Institut für Computersprachen
AB Programmiersprachen und Übersetzerbau

18. Kolloquium

Programmiersprachen und Grundlagen der Programmierung

Bericht 2015-IX-1

Pörtschach am Wörthersee **5.-7. Oktober 2015**

Jens Knoop, M. Anton Ertl (Hrsg.)

Jens Knoop
M. Anton Ertl
Institut für Computersprachen

Technische Universität Wien
`http://www.complang.tuwien.ac.at`

Bericht 2015-IX-1
Schriftenreihe des Instituts für Computersprachen
Technische Universität Wien

Vorwort

Das Kolloquium *Programmiersprachen und Grundlagen der Programmierung* (*KPS*) findet 2015 zum 18. Mal statt. Es setzt eine Reihe von Arbeitstagungen fort, die von den Professoren Friedrich L. Bauer (TU München), Klaus Indermark (RWTH Aachen) und Hans Langmaack (CAU Kiel) ins Leben gerufen wurde. Aus den ursprünglich drei Arbeitsgruppen sind in der Zwischenzeit weitere Forschungsgruppen in ganz Deutschland und darüberhinaus hervorgegangen. Seit 2007 präsentiert sich die Veranstaltung als ein offenes Forum für interessierte deutschsprachige Wissenschaftler. Die folgende Liste gibt einen Überblick über die bisherigen Tagungsorte und Veranstalter und zeigt die inzwischen 35-jährige Tradition der KPS-Treffen:

2013 Lutherstadt Wittenberg	Martin-Luther-Universität Halle-Wittenberg
2011 Schloss Raesfeld, Raesfeld	Westfälische-Wilhelms-Universität Münster
2009 Maria Taferl	TU Wien
2007 Timmendorfer Strand	Universität zu Lübeck
2005 Fischbachau	Ludwig-Maximilians-Universität München
2004 Freiburg-Munzingen	Albert-Ludwigs-Universität Freiburg
2001 Rurberg in der Eifel	RWTH Aachen
1999 Kirchhundem-Heinsberg	FernUniversität in Hagen
1997 Avendorf auf Fehmarn	Christian-Albrechts-Universität zu Kiel
1995 Alt-Reichenau	Universität Passau
1993 Garmisch-Partenkirchen	Universität der Bundeswehr München
1991 Rothenberge bei Steinfurth	Westfälische-Wilhelms-Universität Münster
1989 Hirschegg	Universität Augsburg
1987 Midlum auf Föhr	Christian-Albrechts-Universität zu Kiel
1985 Passau	Universität Passau
1982 Altenahr	RWTH Aachen
1980 Tannenfelde im Naturpark Aukrug	Christian-Albrechts-Universität zu Kiel

Das diesjährige Kolloquium *Programmiersprachen und Grundlagen der Programmierung* (*KPS 2015*) wird nach 2009 zum zweiten Mal vom Institut für Computersprachen der Technischen Universität Wien organisiert. Wir freuen uns, zu diesem 18. Kolloquiumstreffen vom 5. bis 7. Oktober 2015 in Pörtschach am Wörthersee mehr als 70 Teilnehmer von 37 Universitäten, Firmen und außeruniversitären Forschungseinrichtungen aus 10 Ländern begrüßen zu können: aus Österreich, der Schweiz, Deutschland, Schweden, Dänemark, Frankreich, Holland, England, Irland und den USA. Besonders freuen wir uns, Prof. Dr. Dr.h.c. Hans Langmaack als einen der Gründerväter dieser Kolloquiumsreihe unter den Teilnehmern begrüßen zu können sowie auch eine große Zahl Veranstalter früherer Kolloquiumstreffen: Prof. Dr. Gunther Schmidt (1993), Prof. Dr. Tiziana Margaria und Prof. Dr. Bernhard Steffen (1995), apl. Prof. Dr. Thomas Noll (2001), Prof. Dr. Peter Thiemann (2004), Prof. Dr. Clemens Grelck (2007), Prof. Dr. Herbert Kuchen (2011) sowie Dr. Roswitha Picht und Prof. Dr. Wolf Zimmermann (2013).

Ganz besonders freuen wir uns auch, dass nahezu alle Teilnehmer am diesjährigen KPS-Treffen in einem Vortrag über ihre Forschungsarbeit berichten oder diese in Form eines Posters in der erstmalig auf einem KPS-Treffen veranstalteten Poster-Ausstellung vorstellen. Um all dies im Rahmen der zur Verfügung stehenden Zeit zu ermöglichen, verzichten wir in diesem Jahr auf einen eingeladenen Hauptvortrag.

Der vorliegende Tagungsband ist als Bericht 2015-IX-1 in der Schriftenreihe des Instituts für Computersprachen der TU Wien erschienen. Er enthält Ausarbeitungen von 57 in einem Vortrag vorgestellten Beiträgen von mehr als 100 Autoren, zum Teil in Form von Kurzzusammenfassungen. Zusätzlich enthält er Kurzbeschreibungen für einige ausgestellte Poster. Die Themen der Beiträge und Poster zeigen die Breite, Vielfalt und Lebendigkeit der wissenschaftlichen Forschung im Bereich Programmiersprachen und Grundlagen der Programmierung im deutschsprachigen Raum.

Für organisatorische Hilfe und Unterstützung bei der Planung und Vorbereitung dieses Kolloquiums bedanken wir uns sehr herzlich bei der Fachgruppe Programmiersprachen und Rechenkonzepte im Fachbereich Softwaretechnik der Gesellschaft für Informatik (GI), der Oesterreichischen Computer Gesellschaft (OCG), der Einrichtung kinderTUWien und dem Convention Bureau Kärnten. Unser besonderer Dank gilt Ewa Vesely, bei der nicht nur immer wieder alle Fäden der Vorbereitung zusammengelaufen sind, sondern die auch stets entscheidende Ideen in die Planung eingebracht und umgesetzt hat. Herzlich danken möchten wir auch allen Mitarbeitern des Hauses Parkhotel Pörtschach für die gute Zusammenarbeit bei der Vorbereitung dieses Treffens. Ganz besonders bedanken wir uns bei allen Autoren für ihr Engagement und die gute und zeitgerechte Zusammenarbeit, die diesen Tagungsband ermöglicht haben.

Wir wünschen allen Teilnehmern am Kolloquium interessante und spannende Vorträge, fruchtbare Diskussionen und vielfältige Anregungen für die eigene Forschungsarbeit, das Kennenlernen neuer Kollegen und das Wiedersehen guter Bekannter, das Anbahnen und Knüpfen neuer Kontakte und die Vertiefung bestehender Kooperationen, eine kurzweilige und erlebnisreiche Exkursion in die Landeshauptstadt Kärntens Klagenfurt und eine angenehme und stimulierende Zeit in Pörtschach am Wörthersee und der Wörtherseeregion.

Willkommen zur KPS 2015!

Wien, im September 2015

Jens Knoop
M. Anton Ertl

Teilnehmer

Gerald Baumgartner	Louisiana State University, Baton Rouge
Christian Berg	Martin-Luther-Universität Halle-Wittenberg
Annette Bieniusa	Technische Universität Kaiserslautern
Walter Binder	Universitá della Svizzera italiana, Lugano
Dines Bjørner	Danmarks Tekniske Universitet, Lyngby
Stefan Bohne	Technische Universität Ilmenau
Florian Brandner	ENSTA ParisTech, Paris
Stefan Brunthaler	SBA Research, Wien
M. Anton Ertl	Technische Universität Wien
Gerhard Goos	Karlsruher Institut für Technologie
Clemens Grelck	Universiteit van Amsterdam
Matthias Grimmer	Johannes-Kepler-Universität Linz
Fabian Gruber	INRIA - Antenne GIANT, Grenoble
Reiner Hähnle	Technische Universität Darmstadt
Michael Haidl	Westfälische Wilhelms-Universität Münster
Michael Hanus	Christian-Albrechts-Universität zu Kiel
Mathias Hedenborg	Linnæus University, Växjö
Christian Heinlein	Hochschule Aalen - Technik und Wirtschaft
Thomas Heinze	Friedrich-Schiller-Universität Jena
Fritz Henglein	Københavns Universitet
Martin Hentschel	Technische Universität Darmstadt
Stefan Hepp	Technische Universität Wien
Michael Huth	Imperial College London
Matthias Keil	Albert-Ludwigs-Universität Freiburg
Raimund Kirner	University of Hertfordshire
Jens Knoop	Technische Universität Wien
Phillipp Körner	Heinrich-Heine-Universität Düsseldorf
Stefan Kral	Fachhochschule Wiener Neustadt
Andreas Krall	Technische Universität Wien
Philipp Kramer	Hochschule für Technik, Rapperswil
Michael Kruse	INRIA/ENS, Paris
Herbert Kuchen	Westfälische Wilhelms-Universität Münster
Peter Lammich	Technische Universität München
Hans Langmaack	Christian-Albrechts-Universität zu Kiel
Gerald Lüttgen	Otto-Friedrich-Universität Bamberg

Thomas Macht	Universiteit van Amsterdam
Tiziana Margaria	University of Limerick and Lero - The Irish Software Research Centre, Limerick
Christoph Matheja	RWTH Aachen
Alexander Mattes	Technische Universität Kaiserslautern
Eduard Mehofer	Universität Wien
Michael Mendler	Otto-Friedrich-Universität Bamberg
Adriaan Middelkoop	Technische Universität Kaiserslautern
Hans Moritsch	Technische Universität Wien
Ulrich Neumerkel	Technische Universität Wien
Thomas Noll	RWTH Aachen
Viktor Pavlu	Technische Universität Wien
Roswitha Picht	Martin-Luther-Universität Halle-Wittenberg
Richard Plangger	Technische Universität Wien
Martin Plümicke	DHBW Stuttgart
Thomas Prinz	Friedrich-Schiller-Universität Jena
Peter Puschner	Technische Universität Wien
Markus Raab	Technische Universität Wien
Thomas Rupprecht	Otto-Friedrich-Universität Bamberg
Benjamin Saul	Martin-Luther-Universität Halle-Wittenberg
Ursula Scheben	Fachhochschule Dortmund
Doris Schmedding	Technische Universität Dortmund
Gunther Schmidt	Universität der Bundeswehr München
Martin Schoeberl	Danmarks Tekniske Universitet, Lyngby
Dietmar Schreiner	Technische Universität Wien
Michael Schröder	Technische Universität Wien
Sibylle Schwarz	Hochschule für Technik, Wirtschaft und Kultur Leipzig
Steven Smyth	Christian-Albrechts-Universität zu Kiel
Andreas Stadelmeier	DHBW Stuttgart
Bernhard Steffen	Technische Universität Dortmund
Peter Thiemann	Albert-Ludwigs-Universität Freiburg
Baltasar Trancón y Widemann	Technische Universität Ilmenau
Anna Vasileva	Technische Universität Dortmund
Helmut Veith	Technische Universität Wien
Ewa Vesely	Technische Universität Wien
Johannes Waldmann	Hochschule für Technik, Wirtschaft und Kultur Leipzig
Mathias Weber	Technische Universität Kaiserslautern
Mandy Weißbach	Martin-Luther-Universität Halle-Wittenberg
Sebastian Wendt	Martin-Luther-Universität Halle-Wittenberg
Alexander Wenner	Westfälische Wilhelms-Universität Münster
Wolf Zimmermann	Martin-Luther-Universität Halle-Wittenberg
Michael Zolda	University of Hertfordshire

Table of Contents

Default Rules for Curry*

– Extended Abstract –

Sergio Antoy[1] Michael Hanus[2]

[1] Computer Science Dept., Portland State University, Oregon, U.S.A.
antoy@cs.pdx.edu

[2] Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany.
mh@informatik.uni-kiel.de

Abstract. In functional logic programs, rules are applicable independently of textual order, i.e., any rule can potentially be used to evaluate an expression. This is similar to logic languages and opposite to functional languages, e.g., Haskell enforces a strict sequential interpretation of rules. However, in some situations it is convenient to express alternatives by means of compact default rules. Although default rules are often used in functional programs, the non-deterministic nature of functional logic programs does not allow to directly transfer this concept from functional to functional logic languages in a meaningful way. In this paper we propose a new concept of default rules for Curry that supports a programming style similar to functional programming while preserving the core properties of functional logic programming, i.e., completeness, non-determinism, and logic-oriented uses of functions. We discuss the basic concept and sketch an initial implementation of it which exploits advanced features of functional logic languages.

1 Motivation

Functional logic languages combine the most important features of functional and logic programming in a single language (see [3, 6] for recent surveys). In particular, the functional logic language Curry [7] extends Haskell with common features of logic programming, i.e., non-determinism, free variables, and constraint solving. Moreover, the amalgamated features of Curry supports new programming techniques, like deep pattern matching through the use of functional patterns, i.e., evaluable functions at pattern positions [1]. As a simple example, consider an operation `isSet` intended to check whether a given list represents a set, i.e., does not contain duplicates. In Curry, we might implement it as follows ("`++`" denotes the concatenation of lists):

```
isSet (_++[x]++_++[x]++_) = False
isSet _                   = True
```

The first rule uses a functional pattern: it returns `False` if the argument matches a list where two identical elements occur. If this is not the case, the second rule

* This material is based in part upon work supported by the National Science Foundation under Grant No. 1317249.

returns `True`. However, according to the Curry's semantics, *all* rules are tried to evaluate an expression. Therefore, the second rule is always applicable to calls of `isSet` so that the expression `isSet [1,1]` will be evaluated to `False` *and* `True`.

The unindented application of the second rule can be avoided by the additional requirement that this rule should be applied only if no other rule is applicable. We call such a rule a *default rule* and mark it by adding the suffix `'default` to the function's name. Thus, if we define `isSet` with the rules

```
isSet (_++[x]++_++[x]++_) = False
isSet'default _           = True
```

then `isSet [1,1]` evaluates only to `False` and `isSet [0,1]` only to `True`.

In the following, we sketch an implementation of default rules in Curry where we assume familiarity with the basic concepts of functional logic programming and Curry (see [3, 6, 7]).

2 Default Rules

Default rules are often used in both functional and logic programming. For instance, the following Haskell function reverses a two-element list and leaves all other lists unchanged:

```
rev2 [x,y] = [y,x]
rev2 xs = xs
```

The second rule is applied only if the first rule is not applicable, which yields the intended semantics. We can avoid the consideration of rule orderings by replacing the second rule with rules for the patterns not matching the first rule:

```
rev2 [x,y] = [y,x]
rev2 [] = []
rev2 [x] = [x]
rev2 (x:y:z:xs) = x:y:z:xs
```

This coding is cumbersome in general and impossible in conjunction with functional patterns, as used in the first rule of `isSet` above, since a functional pattern conceptually may denote an infinite set of standard patterns (e.g., `[x,x]`, `[x,_,x]`, `[_,x,_,x]`,...). Thus, there is no finite complement of some functional patterns.

In Prolog, one often uses the cut operator to implement the behavior of default rules. For instance, `rev2` can be defined as a Prolog predicate as follows:

```
rev2([X,Y],[Y,X]) :- !.
rev2(Xs,Xs).
```

Although this behaves as intended for instantiated lists, the completeness of logic programming is destroyed by the cut operator. For instance, the goal `rev2([],[])` is provable, but Prolog does not compute the answer {`Xs=[],Ys=[]`} for the goal `rev2(Xs,Ys)`.

These examples show that a new concept of default rules is required for functional logic programming if we want to keep the strong properties of the base language, in particular, the completeness of logic-oriented evaluations. To avoid developing a new logic foundation of functional logic programming with default rules, we try to reuse existing features of functional logic languages. We describe our approach explaining the translation of the default rule for `rev2`. The extension to functional patterns and conditional rules can be done in a similar way.

An operation is defined by a set of "standard" rules and one optional default rule that is applied only if no standard rule is applicable because it do not match or its condition is not satisfiable. For this reason, we translate a default rule into a standard rule by adding the condition that no other rule is applicable. For this purpose, we translate the original non-default rules into "test applicability only" rules where the right-hand side is replaced by a constant (here: the unit value "`()`"):

```
rev2'TEST [x,y] = ()
```

Now we add to the default rule the condition that `rev2'TEST` is not applicable. Since we are interested in the failure of attempts to apply `rev2'TEST`, we use a primitive for encapsulating search to check whether `rev2'TEST` has no value. In functional logic programming, set functions [2] or an operator `allValues` [5] have been proposed for this purpose, which behave similarly to Prolog's `findall` but can be used in a declarative manner. Using these primitives, one could translate the default rule into

```
rev2'DEFAULT xs | isEmpty (allValues (rev2'TEST xs)) = xs
```

Hence, this rule can be applied only if all attempts to apply a non-default rule fail. To complete our example, we add this translated default rule as a further alternative to the non-default rule so that we obtain the definition

```
rev2 [x,y] = [y,x]
rev2 xs | isEmpty (allValues (rev2'TEST xs)) = xs
```

Thanks to the logic features of Curry, one can use this definition also to generate appropriate argument values for `rev2`. For instance, if we evaluate the expression `rev2 xs` with the Curry implementation KiCS2 [4], the search space is finite and computes, among others, the binding {`xs=[]`}. This shows that our concept of default rules is more powerful than existing concepts in functional or logic programming. The actual transformation scheme for default rules is more advanced than sketched above in order to accommodate also functional patterns and conditionals rules and to ensure the optimality of functional logic computations even in the presence of default rules.

3 Examples

To show the advantages of default rules for functional logic programming, we sketch a few more examples. In the classical n-queens puzzle, one must place n queens on a chess board so that no queen can attack another queen. This can be solved by computing some permutation of the list `[1..`n`]`, where the i-th element denotes the row of the queen placed in column i, and check whether this permutation is a safe placement. The latter property can easily be expressed with functional patterns and default rules where the non-default rule fails on a non-safe placement:

```
safe (_++[x]++y++[z]++_) | abs (x-z) == length y + 1 = failed
safe'default xs = xs
```

Hence, a solution can be obtained by computing a safe permutation:

```
queens n = safe (permute [1..n])
```

This example shows that default rules are a convenient way to express negation-as-failure from logic programming. This programming pattern can also be applied to solve the map coloring problem. Our map consists of the states of the Pacific Northwest and a list of adjacent states:

```
data State = WA | OR | ID | BC

adjacent = [(WA,OR),(WA,ID),(WA,BC),(OR,ID),(ID,BC)]
```

Furthermore, we define the available colors and an operation that associates (non-deterministically) some color to a state (the infix operator "?" denotes a non-deterministic choice between its arguments):

```
data Color = Red | Green | Blue

color x = (x, Red ? Green ? Blue)
```

A map coloring can be computed by an operation `solve` that takes the information about potential colorings and adjacent states as arguments, i.e., we compute correct colorings by evaluating the initial expression

```
solve (map color [WA,OR,ID,BC]) adjacent
```

The operation `solve` fails on a coloring where two states have an identical color and are adjacent, otherwise it returns the coloring:

```
solve (_++[(s1,c)]++_++[(s2,c)]++_) (_++[(s1,s2)]++_) = failed
solve'default cs _ = cs
```

References

1. S. Antoy and M. Hanus. Declarative programming with function patterns. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*, pages 6–22. Springer LNCS 3901, 2005.
2. S. Antoy and M. Hanus. Set functions for functional logic programming. In *Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09)*, pages 73–82. ACM Press, 2009.
3. S. Antoy and M. Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, 2010.
4. B. Braßel, M. Hanus, B. Peemöller, and F. Reck. KiCS2: A new compiler from Curry to Haskell. In *Proc. of the 20th Int. Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pages 1–18. Springer LNCS 6816, 2011.
5. J. Christiansen, M. Hanus, F. Reck, and D. Seidel. A semantics for weakly encapsulated search in functional logic programs. In *Proc. of the 15th International Symposium on Principle and Practice of Declarative Programming (PPDP'13)*, pages 49–60. ACM Press, 2013.
6. M. Hanus. Functional logic programming: From theory to Curry. In *Programming Logics - Essays in Memory of Harald Ganzinger*, pages 123–168. Springer LNCS 7797, 2013.
7. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.3). Available at `http://www.curry-language.org`, 2012.

PENCIL: A Platform-Neutral Compute Intermediate Language for Accelerator Programming

Riyadh Baghdadi, Ulysse Beaugnon, Albert Cohen,
Tobias Grosser, Michael Kruse, Chandan Reddy, and
Sven Verdoolaege
INRIA
Email: first.last@inria.fr

Adam Betts, Alastair F. Donaldson, and
Jeroen Ketema
Imperial College London
Email: {a.betts,alastair.donaldson,j.ketema}@imperial.ac.uk

Javed Absar, Sven van Haastregt,
Alexey Kravets, and Anton Lokhmotov[†]
ARM
Email: first.last@arm.com

Róbert Dávid and
Elnar Hajiyev
Realeyes
Email: {robert.david,elnar}@realeyesit.com

***Abstract*—Programming accelerators such as GPUs with low-level APIs and languages such as OpenCL and CUDA is difficult, error-prone, and not performance-portable. Automatic parallelization and domain specific languages (DSLs) have been proposed to hide complexity and regain performance portability. We present PENCIL, a rigorously-defined subset of GNU C99—enriched with additional language constructs—that enables compilers to exploit parallelism and produce highly optimized code when targeting accelerators. PENCIL aims to serve both as a portable implementation language for libraries, and as a target language for DSL compilers.**

We implemented a PENCIL-to-OpenCL backend using a state-of-the-art polyhedral compiler. The polyhedral compiler, extended to handle data-dependent control flow and non-affine array accesses, generates optimized OpenCL code. To demonstrate the potential and *performance portability* of PENCIL and the PENCIL-to-OpenCL compiler, we consider a number of image processing kernels, a set of benchmarks from the Rodinia and SHOC suites, and DSL embedding scenarios for linear algebra (BLAS) and signal processing radar applications (SpearDE), and present experimental results for four GPU platforms: AMD Radeon HD 5670 and R9 285, NVIDIA GTX 470, and ARM Mali-T604.

***Keywords*-automatic optimization; intermediate language; polyhedral model; domain specific languages; OpenCL**

I. INTRODUCTION

Software for hardware accelerators is currently written using low-level APIs and languages such as OpenCL [1] and CUDA [2], which have a steep learning curve, are laborious and error-prone to program with, and lack *performance portability*: the performance of an accelerated application may vary dramatically across platforms. Hence, developing software at this level is unattractive and costly.

A compelling alternative for developers is to program in higher-level languages and to rely on compilers to automatically generate efficient low level code. For general-purpose languages in the C family, this approach is hindered by the difficulty of static analysis in the presence of pointer aliasing. The possibility of aliasing often forces a parallelizing compiler to assume that it is *not* safe to parallelize a region of source code, although aliasing might not actually occur at runtime. Domain-specific languages (DSLs) can help to side-step this problem: it is often clear how parallelism can be exploited given high-level knowledge about standard operations in a particular domain, such as linear algebra [3], image processing [4] or partial differential equations [5]. A drawback of the DSL approach is the significant effort required to implement a compiler generating highly optimized OpenCL or CUDA code for multiple platforms.

To address the above problems, we present PENCIL, a platform-neutral compute intermediate language. PENCIL aims to serve both as a portable implementation language for libraries, and as a target language for DSL compilers.

PENCIL is a rigorously-defined subset of GNU C99 that enforces a set of coding rules predominantly related to restricting the manner in which pointers can be manipulated. These restrictions make PENCIL code "static analysis-friendly": the rules are designed to enable better optimizations and parallelization when translating a PENCIL program to a lower-level program. PENCIL is also equipped with language constructs such as *assume predicates* and *side effect summaries* for functions, which assist with propagating to a PENCIL compiler optimization-enabling information.

PENCIL is easy to learn, as it is C-based. It also interfaces with non-PENCIL C code, which allows legacy applications to be ported incrementally to PENCIL. From the point of view of DSL compilation, PENCIL offers a tractable target: all a DSL-to-PENCIL compiler has to do is to faithfully encode the semantics of the input DSL program into PENCIL—a PENCIL compiler takes care of auto-parallelization and optimization for multiple accelerator targets. Because

[†] anton@dividiti.com

DSL-to-PENCIL compilers have tight control over the code they generate, they can aid the effectiveness of the PENCIL compiler by communicating domain-specific information via the language constructs that PENCIL provides.

We demonstrate the capabilities of PENCIL and its novel static analysis-friendly features in a state-of-the-art polyhedral compilation flow—extended with a PENCIL front-end and implementing advanced combinations of loop and data transfer optimizations. To this end, we consider a number of applications with irregular, data-dependent control and dataflow, making this the first time a fully-automatic polyhedral compilation flow is capable of parallelizing a variety of real-world, non-static-control applications. The applications, which originate from hand-written benchmark suites or were generated by DSL-to-PENCIL compilers, are:

- seven image processing kernels written in PENCIL and covering computationally intensive parts of a computer vision stack used by *Realeyes*, a leader in recognizing facial emotions (http://www.realeyesit.com);
- five benchmarks extracted from the SHOC [6] and Rodinia [7] suites and re-written in PENCIL;
- six kernels generated using the VOBLA linear algebra DSL compiler [3];
- two signal processing radar applications generated from code written in the SpearDE streaming DSL [8].

To assess performance portability, we present an experimental evaluation of generated OpenCL code on four GPU platforms: AMD Radeon HD 5670 and R9 285, Nvidia GTX 470, and ARM Mali-T604. The performance results are promising, considering the implementation efforts for these applications and benchmarks. For example, for the VOBLA linear algebra DSL, we were able to generate code that has performance close to the cuBlas [9] and clMath [10] BLAS libraries [11]. For the Realeyes image processing benchmarks, we could match, and sometimes outperform, the OpenCV image processing library [12].

In summary, our main contributions are:

- PENCIL, a platform-neutral compute intermediate language for direct accelerator programming and DSL compilation;
- a polyhedral compilation flow that leverages the features of PENCIL to handle applications that go beyond the classical restrictions of the polyhedral model, including forms of dynamic, data-dependent control flow and array accesses;
- an evaluation of PENCIL on multiple GPUs and several real-world, non-static-control applications that were previously out of scope for polyhedral compilation.

II. OVERVIEW OF PENCIL

PENCIL is a subset of the C99 language carefully designed to capture static properties essential for implementing advanced loop nest transformations. The language provides constructs that help parallelizing compilers to perform more accurate static analyses and generate efficient target-specific code. The constructs allow communicating information that is difficult for a compiler to extract, but that can be easily captured from DSLs or expressed by expert programmers.

Our aim was for PENCIL to be a strict subset of C99. However, where necessary and when no alternatives existed, we exploited the flexibility of GNU C extensions such as type attributes and pragmas. The pragmas were inspired by familiar annotations for exploiting vector- and thread-level parallelism, but retain a strictly sequential semantics.

PENCIL is not coupled to any particular compiler or target language. However, as we have validated PENCIL using a polyhedral compiler targeting OpenCL, we will refer to this compiler when discussing the implementation of PENCIL.

A. Design Goals

We designed PENCIL with four main goals in mind:

Ease of analysis. The language should simplify static code analysis to enable a high degree of optimization. The main impact of this is that the use of pointers is disallowed, except in specific restricted cases.

Support for domain-specific information. The language should provide facilities that enable a domain expert or a DSL-to-PENCIL compiler to convey domain-specific information that may be exploited by a compiler during optimization. For example, PENCIL should allow the user to indicate bounds on array sizes, enabling placement or staging of arrays in the local memory of a GPU.

Portability. A standard, non-parallelizing C99 compiler supporting GNU C extensions should be able to compile the language. This ensures portability to platforms without specialized PENCIL support and allows existing tools to be used for debugging (unparallelized) PENCIL code.

Sequential semantics. The language should have a sequential semantics to simplify DSL compiler development and direct programming in PENCIL, and, importantly, to avoid committing to any particular parallel patterns.

In designing the PENCIL extensions to C99, we analyzed numerous benchmarks and DSLs [13] and identified language constructs that would be helpful in exposing parallelism and enabling compiler optimizations. In deciding which language features to include, we were guided by the principle that all domain-specific optimizations should be performed at the DSL compiler level, while the PENCIL compiler should be responsible only for parallelization, data locality optimization, loop nest transformations, and mapping to OpenCL. This means that only those properties that are useful for improved static analysis and target mapping need to be expressible in PENCIL. Domain-specific properties that are not useful for optimization do not have to be conveyed and should thus not be a part of PENCIL. This keeps PENCIL general-purpose, sequential and lightweight.

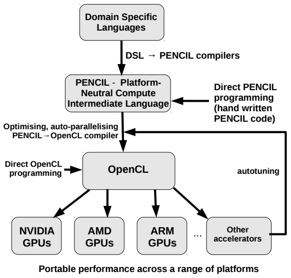

Domain Specific Languages
DSL → PENCIL compilers
PENCIL - Platform-Neutral Compute Intermediate Language
Direct PENCIL programming (hand written PENCIL code)
Optimising, auto-parallelising PENCIL→OpenCL compiler
Direct OpenCL programming
OpenCL
autotuning
NVIDIA GPUs
AMD GPUs
ARM GPUs
...
Other accelerators
Portable performance across a range of platforms

Figure 1. A high level overview of the Pencil compilation flow

Figure 1 gives a high level overview of a typical Pencil usage scenario. First, a program written in a DSL is translated into Pencil. Some domain-specific optimizations may be applied prior to or during this translation, while delaying target-specific optimizations to later compilation stages. Second, the generated Pencil code is combined with hand-written Pencil that implements library functions; Pencil is used here as a standalone language. The combined code is then optimized and parallelized. Finally, highly optimized OpenCL code is generated. The generated code is autotuned through profiling-based iterative compilation.

B. Pencil *Coding Rules*

We detail the most important restrictions imposed by Pencil from the point of view of enabling GPU-oriented compiler optimizations. For more details, see [14], [15].

Pointer restrictions. Pointer declarations and definitions are allowed in Pencil, but pointer manipulation (including arithmetic) is not, except that C99 array references are allowed as arguments of functions. Pointer dereferencing is also not allowed except for accessing C99 arrays. These restrictions essentially eliminate aliasing problems, which is important for parallelization and data movement between the different address spaces of accelerators such as GPUs.

No recursion. Recursive function calls are not allowed, as they are forbidden in languages such as OpenCL.

Sized, non-overlapping arrays. Arrays must be declared using the C99 variable-length array syntax [16], and the declaration of each function argument that is of array type must use `pencil_attributes`, a macro expanding to the C99 `restrict` and `const` type qualifiers followed by the `static` keyword (see Figure 4). During optimization, the Pencil compiler thus knows the length of each array (in parametric form), and knows that arrays do not overlap.

Structured `for` loops. A Pencil `for` loop must have a single iterator, invariant start and stop values, and a constant increment (step), where invariant means that the value does not change in the loop body. Precisely specifying the loop format avoids the need for sophisticated induction variable analyses which may fail under unpredictable conditions.

A further guideline—which is not mandatory as it cannot be statically checked in general—is that array accesses should not be linearized. Linearization obfuscates affine subscript expressions, hindering effective compilation. Multidimensional arrays should be used instead.

Pencil also supports OpenCL scalar builtin functions such as `abs`, `min`, `max`, `sin`, `cos`, using a target-independent and explicitly typed naming scheme (using suffixes to distinguish between `float` and `double` builtins).

C. Assume Predicates

We now describe *assume predicates*, the first main construct introduced by Pencil. The other new constructs—the `independent` directive, summary functions, and the `__pencil_kill` function—follow in Sections II-D–II-F.

An *assume predicate*, written `__pencil_assume(`e`)`, with e a Boolean expression, indicates that e is guaranteed to hold whenever the control flow reaches the predicate. This knowledge is taken on trust by the Pencil compiler, and may enable generation of more efficient code. If e is violated during execution, the semantics of the Pencil program is undefined. This is *not* checked at runtime, but optional runtime checking, for debugging, could be provided. In the context of DSL compilation, an assume predicate allows a DSL-to-Pencil compiler to communicate high level facts.

The *general 2D convolution* example of Figure 2 illustrates the use of `__pencil_assume`. This image processing kernel calculates the weighted sum of the area around each input pixel using a kernel matrix `kern_mat` for the weights. The convolution code is part of an image processing benchmark from Realeyes (see also Section IV-A).

In Realeyes's production environment, the size of the `kern_mat` never exceeds 15×15, as indicated by the assume predicates. While the image processing experts know this, without the predicates the compiler must assume that the kernel matrix can be arbitrarily large. When compiling for a GPU target the compiler must thus either allocate the kernel matrix in the GPU's global memory rather than in fast local memory, or must generate multiple variants—one to handle large kernel matrix sizes and another for smaller kernel matrix sizes—selecting between variants at runtime. Instead, the `__pencil_assume` statements in the code communicate limits on the size of the array, allowing the compiler to store the whole array in local memory.

D. The Independent Directive

The `independent` directive is used as a loop annotation, and is semantically similar to the equally named High Performance Fortran directive [17]. The directive indicates that the result of executing the loop does not depend on the execution order of the data accesses from different loop

```
#define clampi(val, min, max) \
  (val < min) ? (min) : (val > max ) ? (max):(val)

__pencil_assume(ker_mat_rows <= 15);
__pencil_assume(ker_mat_cols <= 15);

for (int i = 0; i < rows; i++)
  for (int j = 0; j < cols; j++) {
    float prod = 0.0f;
    for (int e = 0; e < ker_mat_rows; e++)
      for (int r = 0; r < ker_mat_cols; r++) {
        row = clampi(i+e-ker_mat_rows/2, 0, rows-1);
        col = clampi(j+r-ker_mat_cols/2, 0, cols-1);
        prod += src[row][col] * kern_mat[e][r];
      }
    conv[i][j] = prod;
  }
```

Figure 2. PENCIL code for general 2D convolution

```
/* Examine nodes adjacent to current frontier */
#pragma pencil independent
for (int i = 0; i < n_nodes; i++) {
 if (frontier[i] == 1) {
  frontier[i] = 0;
  /* For each adjacent edge j */
  for (int j = edge_idx[i];
           j < edge_idx[i] + edge_cnt[i]; j++) {
   int dst_node = dst_node_index[j];
   if (visited[dst_node] == 0) {
    /* benign race: threads write same values */
    cost[dst_node] = cost[i] + 1;
    next_frontier[dst_node] = 1;
   }
  }
 }
}
```

Figure 3. PENCIL code fragment for breadth-first search

iterations. As such, the accesses from different iterations may be executed in parallel.

In practice, **independent** is used to indicate that a loop has no loop carried dependences. The directive can also be used when some dependences exist but the user wants to ignore them. In such cases the execution order of the data accesses may have to be constrained using specific synchronization constructs. Examples include reductions implemented via atomic regions, and the use of low-level atomics to give semantics to so-called "benign races", where the same value is written to a location by multiple threads in parallel. It may be necessary to invoke external non-PENCIL functions to enable parallelization of an algorithm that can tolerate arbitrarily-ordered execution of intermediate steps.

The **independent** directive has an effect only on the marked loop, not on any nested or outside loops. It accepts a reduction clause, the purpose of which is to enable parallelization of loops whose only dependences are on variables into reductions are computed. For brevity we do not discuss this clause further.

Figure 3 shows a code fragment of our PENCIL implementation of the breadth-first search benchmark from the Rodinia [7] benchmark suite. The benchmark computes the minimal distance from a given source node to each node in the input graph. The algorithm maintains a frontier and computes the next frontier by examining all unvisited nodes adjacent to the nodes in the current frontier. All nodes in a frontier have the same distance from the source node.

The `for` loop of Figure 3 can be parallelized because each node in the current frontier can be processed independently. This creates a possible race condition on the `cost` and `next_frontier` arrays, but this race condition can be ignored, because all conflicting threads will write the same value. By specifying the **independent** pragma, the programmer guarantees that the race condition is benign, enabling parallelization.

E. Summary Functions

The effect of a function call on its array arguments is usually derived from analyzing the called function. In some cases, the results of such an analysis may be too inaccurate, and in the extreme case, when no code is available, the compiler must conservatively consider the possibility that all elements of each array argument are accessed. To mitigate this problem, PENCIL allows the user to associate a *summary function* with each function. A summary function has a signature identical to the function it is associated with, and the association informs the PENCIL compiler that it may derive the memory accesses from the summary function.

In practice, summary functions are used to describe the memory access patterns of *library functions* called from PENCIL code (and whose source code is usually not available for analysis), and of *non-*PENCIL *functions* called from PENCIL code, as they may be difficult to analyze otherwise. To associate a summary function with a function `foo()`, a programmer uses the attribute **pencil_access**`(name)`, where `name` is the name of summary function describing the accesses of `foo()`.

Summary functions are not executed, but only used for analyzing memory footprints: A summary function must access the same memory elements as the function it is associated with, or an over-approximation thereof. Providing a summary function can enable more precise static analysis than the default conservative assumption that all elements of all array arguments can be accessed. In general, a summary can be simpler than the function it summarizes: it only needs to capture sets of accesses, not their order and number of occurrences. As an example, if a function were to be executed on a processor having no direct access to main memory, the compiler could use its summary to determine the memory elements that would need to be marshaled into and out of the function (cf. [18]).

The functions **__pencil_use** and **__pencil_def** are designed to be used in summary functions to mark memory accesses. A call to **__pencil_use**`(A[`e`])` indicates that a read from array `A` at index e *may* occur, while a call to **__pencil_def**`(A[`e`])` indicates that a write to array `A` at index e *must* occur.

For writes, *may* information can also be conveyed by

```
__attribute__((pencil_access(summary_fft32)))
void fft32(int i, int j, int n,
           float in[pencil_attributes n][n][n]);

int ABF(int n, float in[pencil_attributes n][n][n]) {
 // ...
 for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
   fft32(i, j, n, in);
 // ...
}

void summary_fft32(int i, int j, int n,
           float in[pencil_attributes n][n][n]) {
 for (int k = 0; k < 32; k++)
  __pencil_use(in[i][j][k]);
 for (int k = 0; k < 32; k++)
  __pencil_def(in[i][j][k]);
}
```

Figure 4. Code from Adaptive Beamformer, illustrating summary functions

using a `__pencil_maybe` predicate, which evaluates to a Boolean value unknown at compile-time. More specifically, the conditional

```
if (__pencil_maybe)
 __pencil_def(A[e]);
```

indicates that a write *may* occur to array `A` at index e. This nicely fits any static analysis capable of extracting *may* and/or *must* information from conditional expressions and is also consistent with the usage of wildcards in intermediate verification languages such as Boogie [19].

Figure 4 shows a loop nest extracted from the *Adaptive Beamformer* (ABF) benchmark presented in Section IV-D. The code calls a function `fft32` (a Fast Fourier Transform). The function only reads and modifies (in place) 32 elements of its input array `in`, it does not modify any other parts of the array. The function is not analyzed by the PENCIL compiler because it is not a PENCIL-function. Without a summary function the compiler would conservatively assume that the whole array passed to `fft32` is accessed for reading and writing, preventing parallelization. The summary function indicates that each iteration of the loop nest only reads and writes 32 elements of the input array, allowing the compiler to parallelize the loop nest.

Writing summary functions for library routines is the most common use case for summaries, and is the library developer's responsibility. The summary functions should be provided in the library's header files and are used directly by the PENCIL compiler. In less common cases, summary functions are either written by the PENCIL programmer or automatically generated by a DSL compiler.

F. Kill Statements

The `__pencil_kill` builtin function allows the user to refine dataflow information within and across any control flow region. The `__pencil_kill` function is polymorphic and signals that its argument (a variable or array element) is dead at the program point where the call to the function occurs, meaning that no data flows through this argument from any statement instance executed before the kill to any statement instance executed after.

The information is used in several ways, as explained in detail in [20]. The effect of `__pencil_kill` is illustrated by the following example:

```
__pencil_kill(A);
for (int i = 0; i < n; i++) {
 if (B[i] > 0)
  A[i] = B[i];
}
```

If the above loop is mapped to a GPU kernel, then the `A` array needs to be copied out from the GPU to the host after computation, because some elements of `A` may be written to by the loop. This copy-out overwrites the original contents of `A` on the host. Since not all elements of `A` may be written to, the array must in principle also be copied in to ensure that the elements not written to retain their original values after the copy-out. The `__pencil_kill(A)` statement indicates that the data in `A` is not expected to be preserved by the region and that the copy-in may be omitted.

III. POLYHEDRAL COMPILATION OF PENCIL CODE

We next explain how specific PENCIL features can be compiled with a polyhedral compiler. (But, to reiterate, PENCIL is not tied to any particular compilation technique.)

A. Polyhedral Compilation

Polyhedral compilation uses an abstract mathematical representation to model programs. Each statement in a program is represented using three pieces of information: an *iteration domain*, *access relations* and a *schedule*. The representation is first extracted from the program's AST, it is then analyzed and transformed (loop optimizations are applied during this step), and finally it is converted back into an AST.

The *iteration domain* of a statement is a set that contains all execution instances of the statement (a statement in a loop has an execution instance for each loop iteration upon which it executes). Each execution instance of a statement in a loop nest is uniquely represented by an identifier and a tuple of integers (typically, the values of the outer loop iterators). These integer tuples are compactly described by quasi-affine constraints. For example, the statement on Line 9 of Figure 2, call it $\mathtt{S}_0$, has the following iteration domain:

$$\{\ \mathtt{S}_0\mathtt{(i,j)}\ :\ 0\ \le\ \mathtt{i}\ <\ \mathtt{rows}\ \land\ 0\ \le\ \mathtt{j}\ <\ \mathtt{cols}\ \}$$

A quasi-affine constraint is a constraint over integer values and integer variables involving only the operators +, −, ×, /, %, &&, ||, <, <=, >, >=, ==, !=, and the ternary ?: operator, where the second argument of / and % must be a (positive) integer literal, and where at least one of the arguments of × must be a piece-wise constant expression. An example of a quasi-affine constraint for a statement in a loop nest is $10 \times i + j + n > 0$, where i and j are loop iterators and n is a *symbolic constant* (i.e., a variable that has an unknown but fixed value for the duration of an execution). Examples of non-quasi-affine constraints are $i \times i > 0$ and $n \times i > 0$.

```
if (se[e][r] != 0)
  sup = max(sup, img[cand_row][cand_col]);
```

Figure 5. Code extracted from dilate

```
1 for (int i = 0; i < N; i++)
2   for (int j = 0; j < M; j++)
3     for (int k = 0; k < M; k++) {
4       B[i][j][k] = 0;
5
6       if (A[i][j][k] == 0)
7         break;
8     }
```

Figure 6. Code containing a `break` statement

To be able to extract a polyhedral representation, all loop bounds and conditions need to be quasi-affine with respect to the loop iterators and a fixed set of symbolic constants. This condition is called *static-affine*.

Access relations map statement instances to the array elements that are read or written by those instances, where scalars are treated as zero-dimensional arrays. An accurate representation requires the index expressions in the input program to be static-affine.

Finally, the *schedule* determines the relative execution order of the statement instances. Program transformations are performed via modifications of the schedule and depend on *dependence relations*. These relations map statement instances to statement instances that depend on them for their execution, and are derived from the access relations and the original execution order. In particular, two statement instances depend on each other if they (may) access the same array element, if at least one of those accesses is a write and if the first is executed before the second.

B. Compilation of PENCIL

We adapted `PPCG` [21], an existing polyhedral compiler for GPUs, to handle PENCIL. `PPCG` relies on the `pet` library [22] to extract the iteration domain and access relations; the dependence analysis is performed by the `isl` library [23]. A new schedule is computed by `isl` using a variant of the Pluto algorithm [24] (this latter step applies most loop nest transformations).

We next discuss the changes we made to `PPCG` to support PENCIL. For more details, including details on support for arrays of structures, we refer the reader to [20].

Assume predicates. `pet` keeps track of constraints on the symbolic constants of a program (i.e., of variables that have an unknown but fixed value throughout an execution). The constraints are automatically derived from array declarations and index expressions. In particular, constraints are derived that exclude negative array sizes and negative array indices (negative indices are not allowed because they could result in aliasing within an array). The constraints are used by `PPCG` when generating an AST from a schedule to simplify the generated AST expressions.

An assume predicate provides `pet` with additional constraints on the symbolic constants that may not be automatically derivable. For example, Lines 4, and 5 in Figure 2 provide additional constraints on the symbolic constants `ker_mat_rows` and `ker_mat_cols`. Although the argument of a **`__pencil_assume`** statement can be any expression, `PPCG` currently only exploits quasi-affine ones.

The kill builtin. A kill statement in `pet` represents the fact that no dataflow on the killed data elements can pass through an instance of the statement. This information can be used during dataflow analysis to stop the search for potential sources of data elements. When `pet` comes across a variable declaration, two kill statements that kill the variable are introduced, one at the location of the variable declaration and one at the end of the block that contains the variable declaration. The use of the **`__pencil_kill`** builtin introduces additional kill statements to `pet`.

Non-static-affine array accesses. To handle non-static-affine accesses, `pet` has been modified to distinguish *may*-writes vs. *must*-writes. Any index expression that cannot be statically analyzed or that is not affine, is treated as *possibly* accessing any index. This over-approximation typically results in the compiler statically identifying more dependences than will actually be exhibited at runtime.

Non-static-affine conditionals and loop guards. `PPCG` treats any non-static-affine conditional or loop with a non-static-affine loop guard as a single macro-statement together with its body (i.e., as a statement encapsulating both control and body). Any write inside such a macro-statement is treated as a may-write. For example, the conditional of Figure 5, extracted from the *dilate* benchmark, cannot be analyzed. The if-statement and its body are therefore considered as one macro-statement and the assignment to `sup` is treated as a may-write.

While loops, break and continue. While loops and loops containing break and continue statements are treated like non-static-affine conditionals: the loop and its body are considered to be a single macro-statement. For example, due to the `break` in Figure 6, `PPCG` treats the entire loop headed at Line 3 as a single statement. This means that `PPCG` can schedule (i.e., change the order of execution of) the loop headed at Line 3 and its body as a whole, but it cannot schedule the individual statements in the body.

The independent directive. When the **`independent`** directive is used to annotate a loop, the iterations of that loop may be freely reordered with respect to each other, including reorderings that result in distinct iterations accessing overlapping data. Through the directive the user asserts that no dependences need to be introduced to prevent such reorderings and that any variable declared inside the loop is private to each iteration. `pet` handles the **`independent`** directive by building a relation between the statement instances that excludes them from depending on each other. Moreover, pet builds a set of variables that are local to the loop. This set of variables is used by PPCG to ensure that their live ranges do not overlap in affine transformations, and to privatize them if needed when generating parallel code.

Summary functions. pet has been modified to extract access information from called functions. If a summary function is provided, the information is extracted from the summary instead.

IV. Experimental Evaluation

We evaluated the performance of OpenCL code generated from Pencil using pencilcc, a version of PPCG incorporating a runtime library and the changes discussed in the previous section.[1] To verify that Pencil can be used both as a standalone language and intermediate language for DSL compilers, we used both benchmarks written directly in Pencil and code generated by DSL compilers. The set of benchmarks written directly in Pencil consists of a image processing benchmark suite by Realeyes (Section IV-A) and a selected set of benchmarks from the Rodinia and SHOC suites (Section IV-B). The code generated by DSL-to-Pencil compilers originates from the VOBLA and SpearDE DSLs (Sections IV-C and IV-D).

We used four GPU platforms for our experiments: an Nvidia GTX 470 (with an AMD Opteron Magny-Cours 2×12 core CPU and 16GB RAM), an ARM Mali-T604 (with a dual-core ARM Cortex-A15 CPU and 2GB RAM), an AMD Radeon HD 5670 (with an Intel Core2 Quad Q6700 CPU and 8GB RAM) and an AMD Radeon R9 285 (with an Intel Xeon E5-2640 8 core CPU and 32GB RAM). Hence, we covered both a relatively large set of real-word applications and a relatively diverse range of platforms.

Our experiments were designed to evaluate (a) whether Pencil enables the parallelization (mapping to OpenCL) of kernels that cannot be parallelized with current state-of-the-art polyhedral compilers (Pluto [24]), and (b) whether Pencil enables the generation of efficient code (by comparing the performance of the automatically generated code to hand-crafted code).

Autotuning. We developed an autotuning compiler framework to facilitate the retargeting of our compiler to different GPU architectures. We applied autotuning to the pencilcc-generated code only. Autotuning the hand-crafted reference code (mostly implemented as libraries) would be difficult, because the code is not designed to be autotuned (work group sizes are hard-coded, changing the use of local and private memory requires manual modifications, etc.). Moreover, the BLAS libraries (clMath [10] and cuBlas [9]) do not require autotuning: they are already configured with a set of optimal parameters for their target architectures. Our autotuning framework searches for the most appropriate optimizations (compiler flags) by generating many different code variants and executing them on the target GPUs. The search covers combinations of pencilcc's compiler flags, including different work group and tile sizes, whether to use local and/or private memory, and which loop distribution heuristic to use (out of two possible heuristics). Autotuning each benchmark takes several hours (except for the six VOBLA kernels, which take up to two days due to the large search space).

Measurements. For our experiments, we let pencilcc instrument the generated code to measure the wall clock execution time, which includes the GPU kernel execution time, duration of any data copies (between the host and the GPU), and the time taken to execute on the host any program code that was not offloaded to the GPU. The measured times do not include device initialization and release, and kernel compilation times. In order to exclude compilation time, we either invoked a dry-run computation beforehand that was not timed (caching compiled kernels), or subtracted the compilation time from the total execution time, depending on the way in which the reference implementation compiled and invoked its kernels. We used OpenCL profiling tools to further analyze the performance of the reference implementations and the pencilcc-generated code (obtaining the number of cache misses, device global memory accesses, device occupancy, etc.). Each test was run 30 times. Below, we report the median of the speedups over the reference implementations.

A. Image Processing Benchmark Suite

The image processing benchmark suite consists of a set of kernels covering computationally intensive parts of the computer vision stack by Realeyes ranging from. simple image filters to composite image processing algorithms. For each kernel in the benchmark suite we compared a straightforward (non-hand-optimized) Pencil implementation with the equivalent OpenCL kernel from the OpenCV version 2.4.10 image processing library [12].

The image processing suite consists of 7 kernels: *affine warping*, *image resize*, *general 2D convolution*, *gaussian smoothing*, *color conversion*, *dilate*, and *basic image histogram* (calculating the tonal distribution in an image).

An important characteristic of the image processing kernels is that they contain non-static-affine code, which a classic polyhedral compiler does not handle efficiently due to the restrictions of the polyhedral model. The conditional `if (se[e][r] != 0)` in Figure 5 is an example of such non-static-affine code.

Five kernels from the benchmark suite have non-static-affine conditionals and read accesses. One kernel has non-static-affine write accesses. Hence, the compiler needs to handle all of these. Non-static-affine write accesses are difficult to

[1] Version 0.4 of pencilcc is available at https://github.com/Meinersbur/pencilcc. The experiments in this section were performed using an older, development version: https://github.com/Meinersbur/pencil-driver/tree/7a0dd59708253cb121cadf0b6529bd792b35c3fd.

Table I
EFFECT OF ENABLING SUPPORT FOR INDIVIDUAL PENCIL FEATURES ON THE IMAGE PROCESSING BENCHMARKS

Benchmark	Non-static-affine code	Independent	Assume	Kill
resize	required	-	-	33% ↑
dilate	required	-	-	10% ↑
color conversion	-	-	-	34% ↑
affine warping	required	-	-	23% ↑
2D convolution	required	-	20% ↑	21% ↑
gaussian smoothing	required	-	-	47% ↑
basic histogram	-	required	-	-

Table II
SPEEDUPS OF THE CODE GENERATED BY `PENCILCC` OVER OPENCV FOR THE IMAGE PROCESSING BENCHMARKS

	Nvidia	ARM	AMD Radeon	AMD Radeon
Benchmark	GTX 470	Mali-T604	HD 5670	R9 285
resize	1.00	1.25	2.47	8.09
dilate	0.59	0.32	0.25	2.91
color conversion	1.32	2.37	1.56	1.11
affine warping	1.06	1.93	2.44	2.85
2D convolution	0.91	-	0.95	2.53
gaussian smoothing	0.92	0.97	0.51	1.61
basic histogram	0.45	0.42	0.16	4.34

handle because they prevent the compiler, in general, from determining whether a loop is parallelizable.

The kernels require support for non-static-affine code, the **`independent`** directive, and the **`__pencil_assume`** and **`__pencil_kill`** builtins. Table I lists the features per benchmark. In the case of non-static-affine code and the **`independent`** directive, the table lists whether the feature was required for OpenCL code generation. For the builtins, the table shows the speedup obtained when support for the feature was enabled (vs. disabled). The speedup shown is for the Nvidia GTX 470, the effect on the other platforms was similar. A '-' indicates that a feature was not used in a benchmark or its use did not affect code generation.

Support for non-static-affine code was required to generate OpenCL code for five kernels. For *basic histogram*, the use of the **`independent`** directive enabled parallelization and OpenCL code generation, which is difficult otherwise. For *dilate*, assuming that the size of the structuring element (the array representing the neighborhood used to compute each pixel) is less than 16×16 enabled `pencilcc` to map the element to local memory, and allowed it to generate code that was 20% faster compared to when it did not assume this. The speedups associated with using **`__pencil_kill`** are mainly due to the builtin enabling `pencilcc` to eliminate redundant data copies.

Table II presents the speedups of the `pencilcc`-generated OpenCL code over the baseline OpenCV OpenCL implementation. We used the same image to evaluate all kernels (a 2880×1607, 1.5MB image).

On the AMD Radeon R9 285 platform, the speedup of the `pencilcc`-generated kernels over the OpenCV reference implementations was due to slow data copies used by OpenCV. On this platform, OpenCV used OpenCL's `clEnqueueWriteBufferRect`, which copies data from host to device while at the same time padding the data for aligned memory accesses. `pencilcc`, on the other hand, used OpenCL's `clEnqueueWriteBuffer`, which copies data but does not perform any padding. OpenCV's approach was $7\times$ slower on the AMD Radeon R9 285 platform, explaining the significant speedups we obtain. Note that, although the use of `clEnqueueWriteBufferRect` may be less efficient for these benchmarks, it may be more efficient in other cases where only one data copy is performed and many filters are applied on the same input image.

Other than the difference in data copies, there was no significant difference in the speedups obtained on the AMD Radeon R9 285 HD 5670 platforms, and we focus in the latter AMD platform in the remainder of this section.

`pencilcc` does not apply any optimizations to data copies other than eliminating spurious copies when the user provides appropriate **`__pencil_kill`** statements. For each of the image processing benchmarks, the amount of data copied by the `pencilcc`-generated code (when using **`__pencil_kill`**) was equal to the amount of data copied by the reference implementation. Consequently, the listed speedups (or slowdowns) were solely due to faster (or slower) kernel execution times (except for the R9 285, as discussed above).

The speedups of *resize* and *color conversion* on Nvidia, ARM and AMD Radeon HD 5670 were due to the tiling of the 2D loop nest in these two kernels, which considerably enhanced data locality (up to 56% fewer L1 cache misses on Nvidia for *color conversion*). In the case of *affine warping*, the speedup was due to two optimizations: thread coarsening, which merges multiple work items, leading to less redundant computation, and tiling, which enhanced data locality (up to 65% fewer L1 cache misses on Nvidia).

For *basic histogram*, the code generated by `pencilcc` was generally slower than the OpenCV reference implementation. The OpenCV version was faster, because each work group computes a histogram in local memory, and the local histograms are only combined into one global histogram during a final reduction. Automatic generation of such reductions is not yet supported by `pencilcc`.

In the case of *dilate*, the OpenCV reference implementation was vectorized, while `pencilcc` currently does not support the generation of vectorized code. The lack of vectorization affected the performance most on AMD and ARM. In addition, the OpenCV reference implementation mapped the input image array to local memory while `pencilcc`'s local memory heuristic decided not to apply this mapping. As a consequence, the `pencilcc`-generated code accessed global GPU memory 175 times more than the OpenCV implementation, which led to a decrease in performance. The same problem with the local memory heuristic applied to *gaussian smoothing*.

The performance of *2D convolution* matched that of the OpenCV reference implementation on Nvidia and AMD.

Table III
SELECTED BENCHMARKS FROM THE RODINIA AND SHOC SUITES

Benchmark (Suite)	Data set size	Description/notes
2D stencil (SHOC)	100 iter., 4096 × 4096 grid	On structured grid
Gaus. elim. (Rodinia)	1024 × 1024 matrix	Dense matrix
SRAD (Rodinia)	100 iter., 502 × 458 image	Image enhancement
SpMV (SHOC)	16384 rows	Sparse matrix-vector multipl.
BFS (Rodinia)	4 million nodes	Breadth-first search on graph

Table IV
EFFECT OF ENABLING SUPPORT FOR INDIVIDUAL PENCIL FEATURES ON THE RODINIA AND SHOC BENCHMARKS

Benchmark	Non-static-affine code	Independent
2D stencil	-	-
Gaussian elimination	-	-
SRAD	required	-
SpMV	required	-
BFS	required	required

The reference implementation could not be run on the ARM Mali GPU, as it used hardcoded local memory and work group sizes that exceeded hardware limits.

B. Rodinia and SHOC Benchmark Suites

Our second set of benchmarks consists of reverse-engineered benchmarks from the Rodinia [7] and SHOC [6] suites. We selected the benchmarks, listed in Table III, based on diversity (i.e., covering different Berkeley ‘motifs’ [25] such as dense and sparse linear algebra, structured grids, and graph traversal), and for their ability to pose a challenge to traditional polyhedral compilers due to their use of non-static-affine code. We compared the performance of `pencilcc`-generated code for these benchmarks with the Rodinia and SHOC reference implementations.

Table IV lists the PENCIL features required for each of the benchmarks and shows the effect of the features on `pencilcc`'s ability to generate OpenCL code. Support for non-static-affine code is required by three benchmarks, which use non-static-affine read accesses, conditionals, and write accesses. The non-static-affine write accesses, in *BFS*, prevent the compiler from parallelizing the code, and require use of the **`independent`** directive. We did not make use of **`__pencil_kill`** annotations for the benchmarks in this suite. Assume predicates were useful in providing optimization hints to the compiler for the *2D Stencil*, *SpMV* and *BFS* benchmarks. This was especially important for enabling generation of OpenCL code that could be automatically vectorized by the ARM Mali compiler, but for this benchmark suite we did not conduct a controlled measurement of performance with vs. without assume predicates.

Table V shows the speedups over the OpenCL reference implementations. The speedups for *2D Stencil* and *Gaussian Elimination* are mainly due to tiling which enhanced data locality and reduced cache misses (we observed 4× fewer L1 cache misses for *2D Stencil* on Nvidia GTX 470). For *SRAD*, the PENCIL-generated OpenCL code was significantly slower than the reference implementation, mainly because `pencilcc` did not map a reduction to OpenCL (`pencilcc` currently does not support the generation of parallel reductions). This leads to additional data transfers between the host and the device. For *BFS*, the generated OpenCL code was also slower than the reference code, again due to additional data transfers. These data transfers were due to `pencilcc` only mapping the *bodies* of while loops to the device and generating data transfers at the beginning and end of each loop iteration.

Table V
SPEEDUPS FOR THE OPENCL CODE GENERATED BY `PENCILCC` OVER THE RODINIA AND SHOC REFERENCE IMPLEMENTATIONS

	Nvidia	ARM	AMD Radeon	AMD Radeon
Benchmark	GTX 470	Mali-T604	HD 5670	R9 285
2D stencil	3.44	3.04	2.68	5.76
Gaussian elimination	0.67	1.54	4.39	2.58
SRAD	0.22	0.34	0.43	0.56
SpMV	1.17	1.67	1.04	1.08
BFS	0.65	0.78	0.43	0.72

C. VOBLA DSL for Linear Algebra

The image processing benchmarks and Rodinia and SHOC benchmarks of Sections IV-A and IV-B demonstrate the use of PENCIL as a standalone language. Here and in Section IV-D, we consider benchmarks in which PENCIL is used as an intermediate language for DSL compilers.

VOBLA is a domain specific language for implementing linear algebra algorithms, providing a compact and generic representation using an imperative programming style [3]. The main control flow operators of VOBLA are `if`, `while`, `for`, and `forall`. The `if` and `while` operators have standard semantics. The `for` and `forall` operators iterate over a scalar range (e.g., `0:3`) or arrays. `forall` indicates that the iterations of a loop can be executed in any order.

The VOBLA-to-PENCIL compiler is fairly simple and does not perform any sophisticated optimizations. Advanced loop nest transformations are handled by `pencilcc`. The VOBLA compiler only uses assume predicates and the **`independent`** directive. The **`__pencil_kill`** builtin is only useful to eliminate spurious data transfers in non-static control code and is not needed for the purely static control code of VOBLA. Summary functions are only needed when library functions are called, but these are not generated by the VOBLA compiler.

The VOBLA compiler infers assume predicates from relations between array sizes. For example, for the statement `C = A + B`, the VOBLA compiler infers that the sizes of `A` and `B` are equal and generates a **`__pencil_assume`** statement that indicates this. As a consequence, `pencilcc` does not need to generate code to handle the case in which the sizes of `A` and `B` differ. This information can, e.g., be exploited when `pencilcc` decides to fuse loops that iterate over `A` and `B`, respectively.

The VOBLA compiler generates the **`independent`** directive when translating `forall` operators: each `forall`

Table VI
Performance gains for benchmarks compiled from VOBLA when assume predicates are enabled

Benchmark	Nvidia GTX 470
gemver	6% ↑
2mm	84% ↑
3mm	91% ↑
gemm	71% ↑
atax	13% ↑
gesummv	2% ↑

Table VII
Speedups obtained with `pencilcc` over the BLAS libraries

Benchmark	Nvidia GTX 470	AMD Radeon HD 5670	AMD Radeon R9 285
gemver	1.17	2.14	0.39
2mm	0.91	0.62	0.14
3mm	0.87	0.66	0.12
gemm	1.09	0.69	0.19
atax	0.88	1.79	0.37
gesummv	1.03	1.83	0.33

operator is translated into a Pencil `for` loop that is annotated with **`independent`**.

We used VOBLA to implement a set of linear algebra kernels and compared the code generated by `pencilcc` with equivalent code that calls BLAS library functions. The kernels are *gemver* (vector multiplication and matrix addition), *2mm* (2 matrix multiplications), *3mm* (3 matrix multiplications), *gemm* (general matrix multiplication), *atax* (matrix transpose and vector multiplication), and *gesummv* (scalar, vector and matrix multiplication).

The VOBLA implementations were first compiled to Pencil using the VOBLA compiler and then mapped to OpenCL using `pencilcc`. We compared the code with two highly optimized BLAS library implementations:

- the clMath 2.2.0 [10] BLAS library provided by AMD and used for comparison on the AMD platforms, and
- the cuBlas 5.5 [9] BLAS library provided by Nvidia and used for comparison on the Nvidia platform. In this case we used `pencilcc` to generate CUDA code instead of OpenCL code.

We do not provide a comparison for the ARM platform, as no BLAS library is available on that platform. We used a matrix size of 4096×4096 for all benchmarks.

Table VI shows that the code obtained for the Nvidia GTX 470 was significantly faster with assume predicates enabled. For example, the code generated for *gemm* with assume predicates is 71% faster than without.

Table VII shows the speedups for the kernels generated by `pencilcc` over the BLAS libraries. The `pencilcc`-generated kernels for the Nvidia and the AMD HD 5670 platforms were close in performance to the highly optimized BLAS libraries for *2mm*, *3mm*, *atax* and *gemm* (e.g., 0.69× for *gemm* on the AMD platform). The main optimizations applied to these kernels were tiling, loop fusion, and the use of local and private memory. The BLAS library code still outperforms the `pencilcc`-generated code as it implements many other optimizations such as vectorization (clMath) and the use of register tiling (cuBlas). The speedups for *gesummv* and *gemver* were due to loop fusion and tiling across different BLAS library calls. For example, the *gemver* kernel consists of a sequence of 6 BLAS library calls and although the individual BLAS library functions are highly optimized, better performance can be obtained by fusing and tiling across function calls. clMath is highly vectorized and tuned for the AMD R9 285. Since `pencilcc` does not perform vectorization, it fails to reach the performance levels for clMath on this platform.

D. SpearDE DSL for Data-Streaming Applications

SpearDE [8] is a domain-specific modeling and programming framework for signal processing applications, designed by *Thales Research and Technology*. We evaluated Pencil using two representative SpearDE applications: *Adaptive Beamformer* (*ABF*) and *Space-Time Adaptive Processing* (*STAP*). Both are common signal processing applications for radar systems. We compared the `pencilcc`-generated code with the sequential CPU version, because no parallel version was available to us.

AFB and *STAP* are relatively large: *ABF* consists of 38 statements in the polyhedral representation (with a loop depth reaching five), and *STAP* consists of 88 statements (with a loop depth reaching seven). The *STAP* code is distributed across 12 separate Pencil functions. The functions were optimized individually, because `pencilcc`'s optimization pass currently does not scale to a fully inlined version reaching about 1000 lines of code.

As shown in Table VIII, *ABF* and *STAP* benefit from support for non-static-affine code, the **`independent`** directive, summary functions, and the **`__pencil_kill`** builtin. The speedups reported are again for the Nvidia GTX 470.

As mentioned in section II-E, *ABF* calls a fast Fourier transform function. Without a summary, the compiler assumes that the function modifies its whole input array, making parallelization impossible. The use of the **`independent`** directive in *STAP* enables the parallelization of a loop with non-static-affine array accesses.

Both *ABF* and *STAP* use Pencil only for the computationally intensive parts of the code. Many temporary arrays used in these parts are allocated outside the Pencil regions. However, as `pencilcc` does not analyze non-Pencil code, it cannot assume that the arrays are temporary. Using **`__pencil_kill`** allows the compiler to infer that the arrays do not need to be copied between host and device. In the case of *STAP*, copying the temporary arrays cannot be completely avoided, as the code is distributed across multiple functions and the temporaries are used in several of them.

Table IX shows the speedups of the `pencilcc`-generated code over the sequential code. On all platforms, the speedup for *ABF* was due to parallelization and tiling. The generated code did not make use of local memory, but privatization

Table VIII
EFFECT OF ENABLING SUPPORT FOR INDIVIDUAL PENCIL FEATURES ON THE SPEARDE BENCHMARKS

Benchmark	Summary functions	Non-static-affine	Independent	Kill
ABF	required	required	-	14% ↑
STAP	-	required	6% ↑	4% ↑

Table IX
SPEEDUPS WITH PENCILCC OVER THE SEQUENTIAL CPU CODE FOR THE SPEARDE BENCHMARKS

	Nvidia	ARM	AMD Radeon	AMD Radeon
Benchmark	GTX 470	Mali-T604	HD 5670	R9 285
ABF	11.00	1.88	2.05	3.69
STAP	2.94	0.51	0.89	1.72

of scalars was essential for making parallelization possible. This was also the case for *STAP*, except that the generated kernel code did not perform well on short-vector architectures (ARM Mali and AMD Radeon HD 5670), which suffer from no automatic vectorization in pencilcc.

The performance of *ABF* and *STAP* was also affected by limitations of pencilcc's two loop fusion/distribution heuristics. The first tries to fuse loops as much as possible, which maximizes temporal locality, but does not take into account resource limits (register pressure), resulting in a loss of performance on GPUs. The second heuristic tries to distribute loops as much as possible, which maximizes parallelism but may damage locality (e.g., the imaginary and real parts of complex-valued arithmetic are computed in separate OpenCL kernels when this heuristic is applied). The implementation of a heuristic similar to Pluto's smartfuse heuristic [24] would allow a better trade-off between parallelism and data locality and would enhance performance.

E. Discussion of Results

As our experiments show, the **independent** directive and (in the case of SpearDE) summary functions improve pencilcc's ability to generate OpenCL. Assume predicates and the **__pencil_kill** builtin enhance the quality of the generated code. Performance-wise, 72% of the generated kernels reach at least 50% of the performance of the hand-optimized reference implementations, and in 47% of the cases the generated kernels outperform the reference implementation. Our experiments also expose some limitations of the current setup. In particular, the inability of pencilcc to generate parallel reductions, its limited loop fusion heuristics, handling while loops as black boxes, and the lack of vectorization and register tiling.

We have not discussed the performance of our autotuning framework. In brief: it performed well on small PENCIL benchmarks, but for larger benchmarks (e.g., the SpearDE ones), we ran into problems due a combinatorial explosion in the number of compiler options. This warrants further investigation into search heuristics and predictive modeling.

V. RELATED WORK

Summary functions have first been proposed as abstract domain transformers of numerical libraries in PIPS [26]. As a language construct, they find their origin in the decoupled access/execute (æcute) model [18], which allows expressing memory access patterns and execution constraints of kernels. PENCIL's summary functions are, to the best of our knowledge, the first attempt to abstract interprocedural access patterns in C99.

PENCIL's directives are inspired by directive-based languages such as OpenMP [27] and OpenACC [28]. In PENCIL, the **independent** directive describes the absence of loop carried dependences and such information can be used to enable a range of loop nest transformations rather than enabling loop parallelization alone. A semantically similar directive, also called **independent**, occurs in High Performance Fortran [17]. What sets PENCIL apart is its sequential semantics. As a subset of C, it is designed to allow advanced compilers to perform better static analysis, enabling automatic parallelization.

PENCIL builtins such as **__pencil_assume** allow the PENCIL compiler to receive additional information from a DSL compiler or from an expert programmer. The compiler can exploit this information to enable further optimizations. Microsoft Visual C and clang 3.6 support, respectively, semantically identical **__assume** and **__builtin_assume** builtins. These builtins could be used as a substitute when available.

DSL compilers targeting GPUs typically map DSL code directly to OpenCL and CUDA, relying on DSL constructs that express parallelism. Using such an approach, DSL compilers such as Halide [4] and Diderot [29] (for image processing) and OoLaLa [30] (for linear algebra) show promising results. Our complementary goal is to build a more generic framework and intermediate language to be used by different domain specific optimizers.

Delite [31] is a generic framework for building DSL compilers. Delite relies on information from a DSL to decide whether a loop is parallel but has no facilities for advanced loop nest transformations. We therefore believe that generic DSL frameworks like Delite can benefit from using PENCIL and a polyhedral compiler.

VI. CONCLUSION

We have presented PENCIL, a portable intermediate language designed to enable productive and efficient accelerator programming. PENCIL is unique in its design combining a sequential semantics, strict compliance with C, and a rich set of attributes and pragmas that enable static analysis. PENCIL makes many forms of non-static-affine code and access patterns amenable to advanced loop transformation and parallelization within the polyhedral framework.

We have evaluated the design and implementation of PENCIL on a representative set of benchmarks across several

GPU-accelerated platforms. Some of these benchmarks are written in a domain-specific language and then compiled to PENCIL. Our experiments validate the use of PENCIL together with an optimizing compiler as a valuable building block for enabling performance-portable accelerator programming.

ACKNOWLEDGMENTS

This work was partly supported by the EU FP7 project CARP (287767) and the ARTEMIS COPCAMS project (332913). We would like to thank our partners at Thales Research and Technology for their collaboration on a PENCIL-based interface within the SpearDE framework.

REFERENCES

[1] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in Science & Engineering*, 2010.

[2] Nvidia, "Nvidia CUDA C programming guide 4.0," 2011. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide

[3] U. Beaugnon, A. Kravets, S. van Haastregt, R. Baghdadi, D. Tweed, J. Absar, and A. Lokhmotov, "VOBLA: A vehicle for optimized basic linear algebra," in *LCTES*, 2014, pp. 115–124.

[4] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. P. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *PLDI*, 2013, pp. 519–530.

[5] M. S. Alnæs, A. Logg, K. B. Ølgaard, M. E. Rognes, and G. N. Wells, "Unified form language: A domain-specific language for weak formulations of partial differential equations," *ACM Trans. Math. Softw.*, vol. 40, no. 2, 2014.

[6] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (SHOC) benchmark suite," in *GPGPU*, 2010, pp. 63–74.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*, 2009, pp. 44–54.

[8] E. Lenormand and G. Edelin, "An industrial perspective: A pragmatic high end signal processing design environment at Thales," in *SAMOS*, 2003, pp. 52–57.

[9] Nvidia, *cuBLAS Library User Guide*, 2012. [Online]. Available: http://docs.nvidia.com/cuda/cublas

[10] clMath Developers Team, "OpenCL math library," 2013. [Online]. Available: https://github.com/clMathLibraries

[11] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for Fortran usage," *ACM Trans. Math. Softw.*, 1979.

[12] OpenCV Developers Team, "Open source computer vision library," 2002. [Online]. Available: http://opencv.org

[13] A. Kravets, G. Kouveli, A. Lokhmotov, E. Hajiyev, L. Marak, and T. Virolainen, "CARP deliverable D2.2.A: requirements analysis," 2012. [Online]. Available: http://carp.doc.ic.ac.uk/external/publications/D2.2A.pdf

[14] R. Baghdadi, A. Cohen, T. Grosser, S. Verdoolaege, A. Lokhmotov, J. Absar, S. van Haastregt, A. Kravets, and A. F. Donaldson, "PENCIL language specification," INRIA, Research Rep. RR-8706, 2015. [Online]. Available: https://hal.inria.fr/hal-01154812

[15] R. Baghdadi, A. Cohen, S. Guelton, S. Verdoolaege, J. Inoue, T. Grosser, G. Kouveli, A. Kravets, A. Lokhmotov, C. Nugteren, F. Waters, and A. F. Donaldson, "PENCIL: Towards a platform-neutral compute intermediate language for DSLs," in *WOLFHPC*, 2012.

[16] ISO, "ISO/IEC 9899:1999, Programming languages – C," 1999.

[17] D. B. Loveman, "High performance Fortran," *Parallel & Distributed Technology: Systems & Applications*, vol. 1, no. 1, pp. 25–42, 1993.

[18] L. W. Howes, A. Lokhmotov, A. F. Donaldson, and P. H. J. Kelly, "Deriving efficient data movement from decoupled access/execute specifications," in *HiPEAC*, 2009, pp. 168–182.

[19] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *FMCO*, 2006, pp. 364–387.

[20] S. Verdoolaege, "PENCIL support in pet and PPCG," INRIA, Tech. Rep. RT-457, 2015. [Online]. Available: https://hal.inria.fr/hal-01133962

[21] S. Verdoolaege, J. C. Juega, A. Cohen, J. I. Gómez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for CUDA," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, 2013.

[22] S. Verdoolaege and T. Grosser, "Polyhedral extraction tool," in *IMPACT*, 2012.

[23] S. Verdoolaege, "isl: An integer set library for the polyhedral model," in *ICMS*, vol. 6327, 2010, pp. 299–302.

[24] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *PLDI*, 2008, pp. 101–113.

[25] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from Berkeley," University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, 2006.

[26] F. Irigoin, P. Jouvelot, and R. Triolet, "Semantical interprocedural parallelization: An overview of the PIPS project," in *ICS*, 1991.

[27] OpenMP Architecture Review Board, "OpenMP application program interface, v3.0," 2008.

[28] CAPS Enterprise, Cray Inc., Nvidia, and the Portland Group, "The OpenACC application programming interface, v1.0," 2011.

[29] C. Chiw, G. Kindlmann, J. Reppy, L. Samuels, and N. Seltzer, "Diderot: A parallel DSL for image analysis and visualization," in *PLDI*, 2012.

[30] M. Luján, T. L. Freeman, and J. R. Gurd, "Oolala: An object oriented analysis and design of numerical linear algebra," in *OOPSLA*, 2000, pp. 229–252.

[31] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun, "A domain-specific approach to heterogeneous parallelism," in *PPoPP*, 2011, pp. 35–46.

Typische Muster bei der Implementierung Domänen-spezifischer Sprachen mit Attributgrammatiken*

Christian Berg und Wolf Zimmermann

[1] christian.berg@informatik.uni-halle.de
[2] wolf.zimmermann@informatik.uni-halle.de
Institut für Informatik
Martin-Luther-Universität Halle-Wittenberg

Zusammenfassung. Diese Arbeit präsentiert typische Muster auf Attributgrammatiken und Definitionstabellen. Die präsentierten Muster nutzen, im Gegensatz zu vielen bestehenden Erweiterungen von Attributgrammatiken, Typen und Baumstruktur aus. Analog den Entwurfsmustern Objekt-orientierter Sprachen werden die vorgestellten Muster auf Laufzeiteinfluss, Speichereinfluss und Einsatzgebiet untersucht.

1 Einleitung

Programmiersprachen sowie Domänen-spezifische Sprachen(DSLs[1]) lassen sich mit Attributgrammatiken entwickeln[18,21]. Bei der Entwicklung von Sprachen mit Attributgrammatiken werden ähnliche Attributierungsregeln verwendet. Aufgrund dieser *typischen Muster* existieren eine Reihe von Erweiterungen und Bibliotheken für die Vereinfachung der Spezifikation unter Verwendung von Attributgrammatiken wie bspw. [8,11,25,23,10,16,17,2].

Werden Definitionstabellen und geordnete Attributgrammatiken[12] genutzt, so ist die Auswahl an typischen Mustern eine andere, als dies bspw. bei der Verwendung von Higher-Order Attributgrammatiken der Fall wäre. Letztere erlauben keine Funktionen mit Seiteneffekten und betrachten, im Gegensatz zu geordneten Attributgrammatiken, Attribute nicht als Vor- und Nachbedingungen[11,23]. Analog Entwurfsmustern[9] stellen wir eine Auswahl der typischen Muster in geordneten Attributgrammatiken vor und evaluieren diese bzgl. Laufzeiteinfluss, Speicherverbrauch und Anwendungsfällen.

In Abschnitt 2 stellen wir nochmals kurz Attributgrammatiken und die von uns verwendete Syntax vor. Abschnitt 3 behandelt bestehende und von uns formulierte Muster, welche in Abschnitt 4 evaluiert werden. Ein Überblick über verwandte Arbeiten folgt in Abschnitt 5. Wir geben eine Zusammenfassung und einen Ausblick in Abschnitt 6.

*Diese Arbeit wurde im Rahmen des vom Bundesministerium für Bildung und Forschung (BMBF) geförderten Forschungsprojekts „ELSY“ (Nr. 16M3202D) erstellt.

[1] engl: domain specific languages

$\langle \mathit{Program} \rangle$ ::= $\langle \mathit{Decls} \rangle$

$\langle \mathit{Decls} \rangle$ ::= $\langle \mathit{Decls} \rangle$ $\langle \mathit{Decl} \rangle$
| ε

$\langle \mathit{Decl} \rangle$ ::= $\langle \mathit{VariableDecl} \rangle$

$\langle \mathit{VariableDecl} \rangle$ ::= $\langle \mathit{VarDef} \rangle$ $\langle \mathit{Expression} \rangle$

$\langle \mathit{Decl} \rangle$ ::= $\langle \mathit{Eval} \rangle$

$\langle \mathit{Eval} \rangle$ ::= $\langle \mathit{Expression} \rangle$

$\langle \mathit{Expression} \rangle$ ::= $\langle \mathit{Expression} \rangle$ '+' $\langle \mathit{Expression} \rangle$
| $\langle \mathit{Expression} \rangle$ '-' $\langle \mathit{Expression} \rangle$
| $\langle \mathit{VarReference} \rangle$
| $\langle \mathit{Constant} \rangle$

$\langle \mathit{VarReference} \rangle$ ::= identifier

$\langle \mathit{VarDef} \rangle$ ::= identifier

$\langle \mathit{Constant} \rangle$ ::= number

Abb. 1 – Abstrakte Syntax einer Sprache zur Auswertung von Ausdrücken; Nichtterminale groß beginnend, Terminale klein beginnend

2 Grundlagen

Wir folgen in unserer Präsentation den Darstellungen aus [18], [12] sowie [11]. Eine Kontext-freie Grammatik $\mathtt{G} \triangleq (N, T, P, Z)$ mit Nichtterminalen N, Terminalen T, ausgezeichneter Wurzel $Z \in N$ und Produktionen $p \in P$ definiert eine Sprache $L(\mathtt{G})$. Wird durch $\mathtt{G}$ eine Sprache $L(\mathtt{G})$ definiert, die eine gültige Folge von Grundsymbolen angibt, dann wird $\mathtt{G}$ als *konkrete* Syntax bezeichnet. Definiert $\mathtt{G}$ hingegen Baumaufbaukonstruktoren heißt $\mathtt{G}$ *abstrakte* Syntax. Produktionen werden durch BNF oder EBNF dargestellt und sind eindeutig identifizierbar. Ein Beispiel einer abstrakten Syntax für eine sehr einfache Ausdruckssprache findet sich in Abbildung 1.

Ein Symbol $\mathtt{Y}$ $(\in (N \cup T))$ heißt **ableitbar** aus $\mathtt{X}$, geschrieben als $\mathtt{X} \Rightarrow \mathtt{Y}$, genau dann, wenn eine Produktion mit linker Seite $\mathtt{X}$ existiert, bei der auf der rechten Seite das Symbol $\mathtt{Y}$ vorkommt; formal: $\exists p \in P, p : \mathtt{X} ::= u\ \mathtt{Y}\ \mathtt{v}$ für $u, v \in (N \cup T)^*$. Der reflexiv-transitive Abschluss einer Ableitung wird mit $\overset{*}{\Rightarrow}$ notiert. Üblicherweise werden u und v von uns nicht weiter beachtet und daher auch nicht aufgeführt.

Für eine abstrakte Syntax $\mathtt{G} \triangleq (N, T, P, Z)$ lässt sich nun eine Attributgrammatik $AG \triangleq (\mathtt{G}, A, R, B)$ definieren, wobei A die Menge aller Attribute für alle Terminale und Nichtterminale der abstrakten Syntax $\mathtt{G}$ sind. Für ein Symbol $\mathtt{X} \in N$ mit Attribut $a \in A(\mathtt{X})$ schreiben wir $\mathtt{X}.a$. Für eine Produktion $p \in P$, Symbole $\mathtt{X}_i \in (N \cup T), i \in [1, n]$ und Nichtterminal $\mathtt{X}_0$ schreiben wir p als $p : \mathtt{X}_0$ $\mathtt{::=}\ \mathtt{X}_1\ \cdots\ \mathtt{X}_n$. Solch einer Produktion ist eine Attributierungsregel $r \in R(p)$ zugeordnet; für eine beliebige Funktion f und Symbole der Produktion mit Indizes $j, k, l \in [0, n]$ und beliebige, den Symbolen zugeordnete Attribute $a, b, \ldots, u \in A$, hat r die Form $\mathtt{X}_j.a \leftarrow f(\mathtt{X}_k.b,\ \ldots,\ \mathtt{X}_l.u)$. Analog exisiteren Produktionen zugeordnete Berechnungen B. Ein solches Attribut $\mathtt{X}_j.a$ heißt **definiert** in p; definierte Attribute für ein Symbol auf der linken Seite einer Produktion heißen **synthetisiert**, definierte Attribute für Symbole der rechten Seite einer Produktion **ererbt**. Sind bei Berechnungen oder Attributierungsregeln Infixoperationen üblich, so verwenden wir ebenfalls Infixoperationen, die Identitätsfunktion wird von uns nicht mit aufgeführt.

```
declare_prop val :: Int ← 0

rule Expression ::= Constant
attr Expression.val ← str_to_int(Constant.sym)

rule Expression ::= VarReference
attr Expression.val ← VarReference.key:val

rule Expression_1 ::= Expression_2 '-' Expression_3
attr Expression_1.val ←Expression_2.val - Expression_3.val

rule Expression_1 ::= Expression_2 '+' Expression_3
attr Expression_1.val ←Expression_2.val + Expression_3.val

rule VariableDecl ::= VarDef Expression
attr VarDef.key:val ← Expression.val
   VariableDecl.names_chn ← Expression.names_chn >= VarDef.key:val

chain names_chn
symbol Program
attr head.names_chn ← ∅

symbol VarDef
attr ↑bind ← bind_in_env(↓names_chn, ↑sym)
   ↑key ← keyof(↑bind)
   ↑has_err ← ↑sym ∉ ↓names_chn

   cond ↑sym ∉ ↓names_chn
     => error "Already declared: " ++ ↑sym

symbol VarReference
attr ↑bind ← lookup(↓names_chn, ↑sym)
   ↑key ← keyof(↑bind)
   ↑has_err ← ↑sym ∈ ↓names

   cond ↑sym ∈ ↓names_chn
     => error "Unknown symbol " ++ ↑sym

rule Eval ::= Expression
attr Eval.val ← Expression.val

...

symbol Program
attr cond ↑has_err =>
     error "Cannot evaluate program."
   print(↑val)

bind_in_env :: Env => String => Bind
keyof :: Bind => Key
print :: Int => ()
str_to_int :: String => Int
```

Quelltext 1 – Ausschnitt der Implementierung der Ausdruckssprache mit abstrakter Syntax aus Abb. 1

Wir benutzen in der Praxis geordnete Attributgrammatiken, daher werden Attributierungsregeln und Berechnungen als Vor- und Nachbedingungen aufgefasst[11,12]. Für geordnete Attributgrammatiken lassen sich effiziente Evaluatoren generieren[12]. Eine detaillierte Vorstellung von Attributgrammatiken mit verschiedenen Unterarten findet sich in [3]. Für die Präsentation in dieser Arbeit reicht folgende Definition geordneter Attributgrammatiken aus:

Definition 1. *Eine Attributgrammatik $AG \triangleq (\mathsf{G}, A, R, B)$ mit abstrakter Syntax $\mathsf{G} \triangleq (N, T, P, Z)$ heißt* ***geordnet*** *wenn für jedes Symbol $\mathtt{X} \in (N \cup T)$ und damit assoziierten Attributen $A(\mathtt{X})$ eine Halbordnung existiert, sodass in jedem Kontext von $\mathtt{X}$ die Auswertereihenfolge der Attribute in diesem Kontext diese Halbordnung enthält[12].*

Eine wesentliche Erweiterung, die für die Verwendung typischer Muster notwendig ist, findet sich mit Symbolen und Klassen in Attributgrammatiken.

Definition 2. *Sei $AG \triangleq (\mathsf{G}, A, R, B)$ eine attributierte Grammatik mit abstrakter Syntax $\mathsf{G} \triangleq (T, N, P, Z)$, Nichtterminalen $\mathtt{X}_0 \in N, \mathtt{X}_i \in (N \cup T), i \in [1, n], n \in \mathbb{N}$. Für alle Produktionen $p, q_i \in P$ der Form $p : \mathtt{X}_0{::=}\mathtt{X}_1 \cdots \mathtt{X}_n$, $q_i : \mathtt{X}_i{::=}v_i$, wobei $v_i \in (T \cup N)^*$, den Attributen $a, b \in A(\mathtt{X}_i)$ sowie beliebigen Ausdrücken e_0, e_1 existieren folgende Regelerzeugungsmuster:*

Symbole (u.a. [11,15,19]):

```
symbol X1
attr ↓a ← 10
     ↑b ← f(this.a)
```

entspricht

```
rule q1: X1 ::= v
attr X1.b ←f(X1.a)

rule p: X0 ::= X1 ··· Xn
attr X1.a ←10
```

Klassen (u.a. [11,15,19]):

```
class symbol V
attr ↓a ← 10
     ↑b ← 100
symbol X1 ←V
```

entspricht

```
symbol X1
attr ↓a ← 10
     ↑b ← 100
```

Head und Tail (ebenfalls [11,15,19]):

```
symbol X0
attr head.a ← e0
     ↑b ← tail.b
```

entspricht

```
rule p: X0 ::= X1 ··· Xn
attr X1.a ←e0
     X0.b ←Xn.b
```

Attributierungsregeln werden, abhängig vom Attributtyp (ererbt, synthetisiert), in Symbole bzw. Produktionen übernommen. Wird in einer Produktion ein Attribut eines Symbols bereits definiert, für das eine Symbol-Attributierungsregel existiert, so wird diese nicht übernommen. Analoges gilt für Symbole und Klassensymbole. Zuerst werden Klassensymbol-Attributierungsregeln in Symbole übernommen, dann Symbol-Attributierungsregeln in Produktionen. Wird in mehreren Klassensymbolen ein Attribut definiert und erbt ein Symbol von mehreren dieser Klassensymbole, dann ist die attributierte Grammatik konsistent, genau dann wenn alle vererbenden Klassensymbole dieselbe Attributierungsregel für das definierte Attribut angeben.

Für beliebige Symbole S stehen demnach **`head`** und **`tail`** für das erste bzw. letzte Symbol auf der rechten Seite aller Produktionen mit linker Seite S.

```
symbol X
attr ↑a ← 10
-- Propagation von X.a als Attribut b mit dem Wert von X.a
propagate X.a as b
-- Inklusion des Attributs
symbol Y
attr ↓c ← include X.a
-- Kettenattribut
chain a
```

Quelltext 2 – Erweiterung von Attributgrammatiken mit Inklusion, Propagation, Kettenberechnung und Beträgen, `X` $\overset{*}{\Rightarrow}$ `Y`, Wurzel `Z`[15,7]

Ausgehend von Definition 2 lassen sich Inklusion, Ketten und Unterbaumzugriffe definieren. Eine umfangreiche Behandlung dieser findet sich u.a. in [11,2] sowie [7]. Quelltext 2 zeigt enige dieser existierenden Erweiterungen, wobei der Zugriff auf Attribute in einem höheren Kontext über **`include`** geschieht, **`propagate`** `X.`a **`as`** b analog der Inklusion ein Attribut b mit dem Wert von `X.`a in jedem von `X` aus ableitbaren Nichtterminal definiert und **`chain`** für ein Attribut a eine vorgefertigte Auswertereihenfolge definiert, bei der zuerst von links nach rechts in die Tiefe zur Berechnung abgestiegen wird[15,14]. Diese Auswertereihenfolge kann auch optimiert werden – es ist nicht notwendig in Teilbäume abzusteigen, in denen weder lesender noch schreibender Attributzugriff statt findet, d.h. weder Vor- und Nachbedinungen durch das Kettenattribut in diesem Teilbaum definiert werden[14].

Kommt ein Nichtterminal auf der linken Seite nur in Form von Kettenproduktionen, d.h. Produktionen mit genau einem Symbol auf der rechten Seite der Produktion, vor, dann werden die Kettenattributierungen zum Holen eines synthetisierten Attributs von uns nicht mit aufgeschrieben (Lösungsansätze dafür werden u.a. in [22,15,14] vorgestellt). Wie in [10] und [11] beschrieben, können diese Attributierungen aus einer Spezifikation heraus generiert werden. Bei Nichtterminalen aus denen nur Terminale direkt ableitbar sind, wird auf den Wert, bzw. die Zeichenfolge, des Terminalsymbols mit dem Attribut `sym` von diesem Nichtterminal aus zugegriffen.

Die Typisierung der Funktionen, Attribute, Bedingungen und Ausdrücken folgt typischerweise der Programmiersprache, für die ein Evaluator generiert wird oder in der der Evaluator interpretiert wird. Für die Präsentation in dieser Arbeit wird ein an Haskell angelehntes Typsystem ohne Klassen, Monaden oder parametrischer Datentypen, aber traditioneller mathematischer Funktionsapplikation, genutzt. Funktionen mit Seiteneffekt werden von uns statt mit Monaden mit dem Typ `()` deklariert. Dieser „Typ“ kann nur als letzter Typ, oder im Falle von Funktionen, die Speicher allozieren, als erster Argumenttyp vorkommen. Typen werden, abzüglich eben genannter, wie in Haskell definiert. Wir verwenden die Listendarstellung Haskells zur Beschreibung allgemeiner mehrelementiger Datentypen, bspw. Listen, Arrays oder Mengen.

Einhergehend mit den Konventionen von Haskell werden Terminale, Nichtterminale, Typen und Typkonstruktoren bei uns am Anfang groß geschrieben; Variablen, Funktionsnamen und Attribute hingegen werden am Anfang klein geschrieben. Funktionen werden bei uns wie in Haskell typisiert, wenngleich wir Currying in der Praxis nicht einsetzen.

Die üblichen Datentypen wie Bool, Int, String und Typen zur Code-Generierung und Namensanalyse sind vordefiniert. Zum Zugriff auf die Definitionstabelle dient der Typ Key, zum Aufbau des Namensraums werden Env und Bind genutzt.

Die Definitionstabelle, als Bestandteil der Namensanalyse, kann automatisch aus Spezifikationen generiert werden[25]. Wir beschreiben eine Eigenschaft n, die in der Definitionstabelle abgelegt werden soll, mit

```
declare_prop n :: τ ←e
```

oder

```
declare_prop n :: τ
```

wobei τ ein beliebiger Typ ist und e ein beliebiger Ausdruck. Letzterer gibt an, welcher Wert angenommen werden soll, wenn ein Eintrag nicht in der Definitionstabelle vorhanden ist. Das Setzen eines Wertes in der Definitionstabelle erfolgt über ein Attribut vom Typ Key gefolgt von einem Doppelpunkt und dem Namen der zu setzenden Eigenschaft. Soweit möglich, verzichten wir auf die Präsentation der Typen, wenn diese aufgrund der Verwendung hergeleitet werden können oder für das Verständnis bzw. das vorgestellte Muster nicht von Relevanz sind.

Zusätzliche Abhängigkeiten, die in geordneten Attributgrammatiken (siehe [12]) notwendig sein können, werden von uns mit

```
symbol X
attr ↑gotB ← true >= constituent Y.gotB
```

definiert, wobei >= für die zusätzliche Abhängigkeit einer Attributierung steht und **constituent** alle von X ableitbaren Y „aufsammelt“. Bei der Definition eines Wertes kommt in unserer Präsentation **constituent** nur vor, wenn der Wert irrelevant ist und dadurch nur eine Abhängigkeit geschaffen werden soll. Für eine detaillierte Einführung zu **constituent** wird auf [15] und [11] verwiesen.

Quelltext 1 zeigt einen Ausschnitt zur Implementierung der Ausdruckssprache aus Abbildung 1 unter Verwendung der bisher eingeführten Notation und Muster.

3 Typische Muster und deren Äquivalenzen

Bei der Vorstellung der Muster betrachten wir eine abstrakte Syntax definiert durch eine Grammatik $\mathsf{G} \triangleq (T, N, P, Z)$ mit attributierter Grammatik $AG \triangleq (\mathsf{G}, A, R, B)$. Für die folgende Präsentation sei folgende Notation eingeführt: für $i \in [0, n], j \in [0, m], m, n \in \mathbb{N}$ sind als Nichtterminale $\mathtt{A}, \mathtt{B}, \mathtt{X}_i \in N$ sowie als

Terminale $E_j \in T$ üblich. Mit kleinen Buchstaben werden Attribute $a, b, \cdots, z \in A$ notiert. Wir fordern üblicherweise, dass $Z \overset{*}{\Rightarrow} \mathtt{A}$, sowie für alle $i \in [0, n], n \in \mathbb{N} : \mathtt{A} \overset{*}{\Rightarrow} \mathtt{X_i}$ gilt[2]. Werden in den aufgelösten Mustern Attribute verwendet, die bei der Definition des Musters nicht verwendet werden, so sind dies neue, bisher nicht benutzte oder definierte, Attribute. Werden bei den folgenden Mustern keine Quellen in der Definition angeben, so konnten dazu von uns bisher keine Quellen gefunden werden, die das Muster so oder in ähnlicher Form beschreiben.

3.1 Direkte Muster, Basismuster und Kombinationen

Ein häufig auftretendes Muster, bspw. bei der *Hashwert-Berechnung* oder *Codegenerierung*, ist die Bestimmung eines Attributs auf Basis eines anderen Attributs.

Definition 3. ***(Attributabbildung)***

```
a is f_x (X0.b)
```

entspricht

```
symbol X0
attr ↑a ←f_x(this.b)
```

für ein $\mathtt{X}_0$ *für das b definiert wird und einem neuen Attribut a. Wird* $\mathtt{X}_0$*. vor b weggelassen, so gilt diese Abbildung für jedes Symbol* $\mathtt{X}_i$*, für das b definiert wird.*

Diese *Attributabbildung* findet sich z.B. in Quelltext 1 bei der Berechnung von `VarDef.key` aus dem Attribut `bind`. Der zweite Teil der Definition erlaubt zusätzlich die Berechnung von `VarReference.key` ohne weitere Spezifikation. Es wird eine Vereinfachung der Attributabbildung für mehrere Symbole der abstrakten Syntax ermöglicht. In anderen Fällen kann man die Attributabbildung heranziehen um andere Muster zu beschreiben. Aufgrund der einfacheren Präsentation und der Möglichkeit der verlustfreien Abbildung von Baumstrukturen auf Listen und wieder zurück[24] betrachten wir bei den Mustern diese als Liste von Attributwerten. Unter dieser Betrachtung beschreibt Def. 3 die Funktion `map` $:: (\tau_a \to \tau_b) \to [\tau_a] \to [\tau_b]$ bzw. den einzelnen Schritt von `map`.

Ebenfalls häufig bei der *Codegenerierung* oder *Typisierung* anzutreffen sind einfache Beiträge. Diese werden genutzt um eine Faltung über Attribute von Knoten des abstrakten Syntaxbaumes durchzuführen.

Definition 4. ***(einfache Beiträge****, u.a. [7,2,11])*

```
contribute X1.a, ···, Xn.o
  to A.b ←e using ⊕
```

entspricht

```
chain b
symbol A
attr head.b ← e

symbol X1
attr ↑b ← tail.b ⊕ this.a
···
symbol Xn
attr ↑b ← tail.b ⊕ this.o
```

für eine binäre Operation $\oplus\colon \tau \to \tau \to \tau$ *und einen Initialisierungsausdruck e, der auch durch eine Konstante c repräsentiert werden kann.*

[2] Damit gilt auch $Z \overset{*}{\Rightarrow} \mathtt{X}_i$ für $i \in [0, n], n \in \mathbb{N}$.

Für einen Typ τ und eine binäre Operation $\oplus :: \tau \to \tau \to \tau$ sowie eine Konstante c muss $(\tau, \oplus, c)$ ein Monoid sein, d.h. c ist das neutrale Element von τ bzgl. der Operation $\oplus$, welche assoziativ sein muss. Bei der Betrachtung als Liste von Attributen sind einfache Beiträge der funktionalen Programmierung mit `fold` nicht fern. Allerdings unterscheidet sich bei einfachen Beiträgen die Signatur mit `fold` $:: (\tau \to \tau \to \tau) \to \tau \to [\tau] \to \tau$ geringfügig von der bekannten Signatur funktionaler Programmierung.

Ist die Operation $\oplus$ nicht assoziativ können Umsortierungen und andere Aktionen beim Aufbau der abstrakten Syntax (siehe dazu auch [10]) zu unerwarteten Ergebnissen führen. Die Auswertung folgt der Baumstruktur des abstrakten Syntaxbaums.

Einfache Beiträge und Attributabbildungen können genutzt werden um das *Aufsammeln* von Attributen in ein anderes Attribut oder gar einen Definitionstabelleneintrag zu erreichen[14]. Dies ist nicht zu verwechseln mit *Collection*-Attributen aus [7,2], denn diese sind bereits in Beiträgen (Def. 4) gebündelt. Das Aufsammeln von Attributen kann in der Codegenerierung und auch Typisierung genutzt werden um Funktionsparameter oder auch Parametertypen aufzusammeln.

Definition 5. *(**Aufsammeln**, u.a. in [2,7])*

```
collect X₁.b,···,Xₙ.b in X₀.a
```

entspricht

```
X₁.b' is (X₁.b:[])
...
Xₘ.b' is (Xₘ.b:[])
contribute X₁.b', ··· Xₘ.b'
  to X.a ← [] using ++
```

wobei $(:)\colon \tau_Y \to [\tau_Y] \to [\tau_Y]$ *ein Listenkonstruktor,* $[]$ *die leere Liste darstellt,* $++\colon [\tau_Y] \to [\tau_Y] \to [\tau_Y]$ *die Listenkonkatenation und* τ_Y *der Typ der Attribute* $\mathtt{X}_i.b$ *ist; die aufgesammelten Attribute müssen denselben Typ haben.*

Definition 5 beschreibt somit eine Parametrierung einfacher Beiträge um Attribute aufzusammeln, wobei eine wesentlich kompaktere Darstellung erreicht wird, als dies mit Beiträgen selbst erreichbar ist. Nicht nur Funktionsparameter und Parametertypen lassen sich dadurch aufsammeln, sondern ebenso Konstruktoren oder Alternativen. Das Aufsammeln beschreibt also die Anwendung der Faltung um die implizite Liste der Baumstruktur explizit in einem Attribut abzulegen.

Ein wiederkehrendes Muster, bspw. *bei* der *Lebendigkeitsanalyse* oder der Analyse *allgemeiner Abhängigkeitsbeziehungen*, ist das „runterreichen, zusammenfassen, aufsammeln und ablegen“. Beispielhaft für dieses Muster, da es im Rahmen der Analyse von Pumpensystemen (siehe [1]) häufig vorkommt ist die Beschreibung von Abhängigkeitstypen.

Definition 6. *(**Abhängigkeitsaufbau**) Seien* τ_1 *und* τ_2 *Typen*

```
deptype of (τ₁, τ₂) is
  A.a => X₀.b in B.c
```

entspricht

```
propagate A.a as a'
X₀.b' is ((X₀.a', X₀.b):[])
contribute X₀.b'
  to A.sub_c ← [] using ++
contribute A.sub_c
  to B.c ← [] using ++
```

wobei $B \overset{*}{\Rightarrow} A$ und der Typ von `A.a` *bzw.* $\mathtt{X}_0$`.b` τ_1 *bzw* τ_2 *ist.*

Der Abhängigkeitsaufbau wird ausschließlich durch Rückführung auf andere Muster erreicht.

Soll die Sortierung von Elementen bestimmt werden, Listen durchnummieriert werden oder ganz allgemein etwas unter Definition weiterer Attribute bestimmt werden, so nutzen wir dafür komplexe Beiträge. Ein komplexer Beitrag erweitert einen einfachen Beitrag um Zwischenergebnisse und zusätzlicher Verallgemeinerung.

Definition 7. *(**komplexer Beitrag**) Für Attribute und Definitionstabelleneigenschaften* $\mathtt{x}_i$*,* $i \in [1, n]$*, Ausdrücke* $\mathtt{e}_j$ *für* $j \in [1, p], p \in \mathbb{N}$ *ist*

```
contribute X1.a, ···, Xn.o
  to A.c ←e
via x1 ←e1,
     ···
     xo ←eo,
     chain ← ep
```

äquivalent mit

```
chain c
symbol A
attr head.c ← e

symbol X1
attr ↑xo ←eo
    ···
    ↑x1 ←e1
    ↑c ←ep
      >= (t.xo, ···, t.x1)
···
symbol Xn
attr ↑xo ←eo
    ···
    ↑x1 ←e1
    ↑c ←ep
      >= (t.xo, ···, t.x1)
```

Bei komplexen Beiträgen erfolgt der Beitrag erst nach Berechnung aller **`via`**-Attribute. Diese **`via`**-Attribute können als Seiteneffekte während der Faltung betrachtet werden. Folgende Annahme muss in weiteren Arbeiten überprüft werden.

Hypothese 1. (**Ordnungserhaltend**): Für eine gegebene geordnete Attributgrammatik ist diese durch Hinzufügen der Expansion der Definitionen 4 und 7 weiterhin geordnet (Def. 1).

Durch komplexe Beiträge kann der Quelltext wesentlich knapper dargestellt werden, jedoch ist Hauptanwendungsfall die Umsetzung anderer Muster und Implementierung von Modulen.

Ein solches Muster ist die Akkumulation von Werten und stellt eine Parametrierung komplexer Beiträge dar. Für ein Symbol bestimmt die *Präfixsumme* eine Akkumulation eines Attributs in der Reihenfolge der Eingabe und legt Zwischenergebnisse in einem Attribut ab. Diese werden als Liste in einem Attribut zur Verfügung gestellt.

Definition 8. *(**Präfixsumme**) Seien die Attribute* $\mathtt{X}_1$`.a`, $\cdots$ $\mathtt{X}_n$`.j` *und eine (assoziative) binäre Operation* $\oplus :: \tau \to \tau \to \tau$ *dann sind*

```
scan X1.a, ···, Xn.j
  to A.x ← c with y using ⊕
```

äquivalent

```
contribute X1.a, ···, Xn.j
  to A.chn ← c
  via y ←chain ⊕ tribute
  chain ← chain ⊕ tribute
collect X1.y,···,Xn.y
  in A.x
```

wobei **`tribute`** *die Referenz auf den aktuellen Beitrag ist.*

Für Präfixsummen bilden $(\tau, \oplus, c)$ wieder einen Monoid.

Wie bereits beschrieben kommt es ebenfalls häufig vor Indizes zu bestimmen. Da dies bei uns sehr häufig vorkommt, bspw. bei der Codegenerierung für die Zugriffsauswahl bei Feldern und Verbünden, existiert folgendes Muster.

Definition 9. ***(Indexbestimmung)***

```
count X0, ···, Xn
  from A in s start with e
```

entspricht

```
symbol X0
attr ↑trib ← 1

···

symbol Xn
attr ↑trib ← 1
scan X0.trib, ··· Xn.trib
  to A.cnt ← e
  with s using +
```

wobei `cnt`, `trib` *und s neue Attribute sind und e ein Initialisierungswert.*

Neben der beschriebenen Äquivalenz zur Indexbestimmung könnten auch komplexe Beiträge ohne Umweg herangezogen werden.

Komplexe Beiträge sind somit vielseitig nutzbar. Ein weiteres Beispiel dafür zeigt sich darin, dass diese genutzt werden können um die Namensanalyse, ähnlich wie [16], umzusetzen. Im Gegensatz zu der Variante aus [16] ist die folgende Definition nicht über Bibliothekscode und Vererbung oder reiner Textersetzung umgesetzt. Bei der Namensanalyse kommt es sehr oft vor, dass bereits ein umfangreicher Teil der Definitionstabelle gefüllt werden kann.

Definition 10. ***(Namensanalyse)***

```
use_before_def X1.sym, X2.sym
  in A.n props: x1 ⊖ C.a,
                ···
                xo ⊖ D.y
  with error unknown,
       error unique
```

entspricht

```
contribute X1.sym to A.n
via x1 ←X1.c_a
  ···
  xo ←X1.d_y
  bind ← nbnd(chain,tribute)
  key ← keyof(bind)
  chain ← chain
propagage A.n as env

symbol X2
attr cond ↑sym ∈ this.env
  => error "not defined"

symbol X1
attr cond ↑sym ∉ ↓n
  => error "already defined"
```

wobei $\mathtt{C}, \mathtt{D} \in (N \cup T)$, $\mathtt{x}_i$, $i \in [1, n], n \in \mathbb{N}$ *Attribute und Definitionstabelleneigenschaften,* $\mathtt{X}_1 \overset{*}{\Rightarrow} \mathtt{D}$ *und* $\mathtt{X}_1 \overset{*}{\Rightarrow} \mathtt{C}$. *Seien* τ *der Typ eines Attributs* y *eines aus* $\mathtt{X}_1$ *ableitbaren Symbols* $\mathtt{X}_i$, *analog* $\mathtt{D}.y$ *und* $\mathtt{C}.a$, *entspricht* **`props:`** $\mathtt{x}_a \ominus \mathtt{X}_i.y$

```
collect X_i.y in X_1.c_a
```

bzw.

```
contribute X_i.y
  to X_1.c_a ←c using r
```

wenn der Typ von $\mathtt{X}_1.\mathtt{x}_a$ $[\tau]$ ist (linke Seite), respektive in allen anderen Fällen; wobei (τ, r, c) einen Monoid bilden. Die Attribute `bind` *und* `key` *sind neue, noch nicht verwendete Attribute zum Zugriff auf die Definitionstabelle. Dazugehörige Funktionen erstellen einen Eintrag im Namensraum (*`nbnd`*) und machen diesen als Definitionstabelleneintrag (*`keyof`*) verfügbar.*

Eine weitere zu prüfende Eigenschaft ist das Verhalten der Kombination dieser Muster miteinander. Seien A_m, A_n und A_e die Attribute eines Musters, wobei A_m die referenzierten Attribute eines Muster darstellt, A_n die neuen Attribute bei der Expansion eines Musters und $A_e \triangleq A_m \cup A_n$ die expandierten Attribute eines Musters sind und $A_m \cap A_n = \emptyset$ gilt. Weiterhin seien zwei beliebige Muster m_1 und m_2, wobei $A_{n_{m_1}} \cap A_{n_{m_2}} = \emptyset$ und eine geordnete Attributgrammatik $AG \triangleq (\mathsf{G}, A, R, B)$. Folgende Annahme muss von uns noch bewiesen werden:

Hypothese 2. (**Abgeschlossenheit der Kombination**) Für zwei beliebige Muster m_1 und m_2 und expandierter Varianten $M_1 \triangleq R_{m_1} \cup B_{m_1}$, $M_2 \triangleq R_{m_2} \cup B_{m_2}$ ist $AG' \triangleq (\mathsf{G}, A \cup A_{n_{m_1}} \cup A_{n_{m_2}}, R \cup R_{m_1} \cup R_{m_2}, B \cup B_{m_1} \cup B_{m_2})$ eine geordnete Attributgrammatik.

3.2 Muster der Definitionstabelle

Bei Verwendung der Definitionstabelle unter Beachtung der Ordnungseigenschaft geordneter Attributgrammatiken treten weitere Muster auf. Zum Befüllen und dann Abfragen, bspw. bei der Alias- oder Typanalyse, kommt folgendes Muster zum Einsatz:

Definition 11. ***(Speichern und Laden):***

```
store_load X_0.x_1,···,X_n.x_n
  in A.a with A.k:p :: τ
via Y_1.b_1 ←e_1
     ...
     Y_m.b_m ←e_m
```

bzw.

```
declare_prop p :: τ
symbol S
attr ↑gotStA_k_p ← true
     >= constituent A.k:p
symbol Y_1
attr this.b_1 ←e_1
     >= include S.gotStA_k_p
...
symbol Y_m
attr this.b_m ←e_m
     >= include S.gotStA_k_p
symbol A
attr this.a ← A.k:p
     >= include S.gotStA_k_p
```

wobei $\mathtt{Y}_j \in (N \cup T), j \in [1, m], m \in \mathbb{N}$; $S \triangleq Z$, und die $\mathtt{X}_i.\mathtt{x}_i, i \in [0, n]$ analog Def. 10 über Aufsammeln oder Beitrag in die Definitionstabelleneigenschaft $\mathtt{A.k} : \mathtt{p}$ *hinzugenommen wird. Die* ***`via`****-Attribute müssen nicht aufgeführt werden.*

Das Muster zum Speichern und Laden stellt also sicher, dass, bevor eine Spalte der Definitionstabelle geladen wird, der Eintrag in dieser Spalte vorhanden ist.

Für viele der Muster aus Abschnitt 3.1 existieren Varianten, die die gewonnenen Informationen in die Definitionstabelle überführen. Diese Varianten unterscheiden sich nur marginal – durch Verwendung von zu speichernden Eigenschaften statt Attributen – von den ursprünglichen Mustern. Wir verzichten daher auf einer Präsentation dieser Muster.

3.3 Weitere Muster

Viele Muster lassen sich durch Bibliotheken umsetzen. Beispiele dafür sind die Namensanalyse oder die Analyse der Gültigkeitsbereichze von Bezeichnern (Scoping)[16]. Andere Muster basieren auf der der Generierung von Konstruktoren, die in der Attributgrammatik als Makros zur Verfügung gestellt werden können, bspw. Codegenerierung oder Typanalysen[17,25]. Eine genaue Vorstellung dieser Ansätze würde den Rahmen dieser Arbeit sprengen.

Muster, die die Definitionstabelle analysieren oder traversieren existieren ebenso wie Muster, die Einträge in der Definitionstabelle aus bestehenden Einträgen bestimmen. Die Beschreibung der Traversierung der Definitionstabelle kann ebenso kompakt erfolgen, wie OCL dies für andere Mengen-orientierte Datentypen erreicht.

4 Evaluierung der Muster

Anhand des Sprachbeispiels aus Abschnitt 2, insbesondere Quelltext 1, wird vorgestellt, wie sich die Muster bezüglich Speicher- und Laufzeiteinfluss verhalten.

Die Verwendung von Mustern statt der Quellen aus Quelltext 1 zeigt Quelltext 3.

Das konkrete Speicher- und Laufzeitverhalten der Muster ist abhängig von der konkreten Ausprägung des Musters und der verwendeten Hilfsfunktionen sowie Algorithmen. Darüber hinaus ist auch die verwendete Grammatik sowie die Eingabe an den generierten Evaluator oder Übersetzer wichtig.

Wir beschränken uns daher darauf nur auf einige Sonderfälle einzugehen. Eine Propagation erzeugt, wenn das Attribut aufgrund von Optimierungen eine globale Variable ist (siehe dazu [13]), ein neues Attribut. Das Erlauben von Seiteneffekten könnte andernfalls zu Änderungen von Werten zwischen Start der Propagation und Auswertung des propagierten Attributs an anderen Stellen führen. Weiterhin erzeugen **`via`**-Attribute neue Attribute bzw. Definitionstabellenspalten.

Die Laufzeit der Muster ist selten höher als ein kompletter Durchlauf des abstrakten Syntaxbaums. Eine Ausnahme bilden die auf komplexen Beiträgen aufbauenden Muster sowie Muster der Definitionstabelle, welche einen zweiten Besuch eines Knotens nach sich ziehen. Diese Betrachtung schließt jedoch nicht aus, dass es keine zusätzlichen Besuche eines Knotens gibt, wenn andere Attributierungsregeln vorhanden sind.

```
declare_prop val :: Int ← 0
val is str_to_int(Constant.sym)
def_before_use VarDef.sym, VarReference.sym in Program.names_chn
  with error unknown, error unique

symbol VarDef
attr ↑has_err ← ↑sym ∉ ↓names_chn

symbol VarReference
attr ↑has_err ← ↑sym ∈ ↓names

...

deptype of (Key, Key) is VarDef.key =>VarReference.key in Program.deps
count VarDef from Program in defs

symbol Program
attr ↑has_err ← cyclic(↑deps) || Program.defs > 1
   cond ↑has_err =>
     error "Cannot evaluate program."
   print(↑val)
```

Quelltext 3 – Ausschnitt der Implementierung der Ausdruckssprache mit abstrakter Syntax aus Abb. 1

5 Verwandte Arbeiten

Attributgrammatiken wurden von Knuth in [18] eingeführt. Wesentliche Erweiterungen der klassischen Attributgrammatiken finden sich in geordneten Attributgrammatiken (Ordered Attribute Grammars) von Kastens beschrieben in [12], wiederbeschreibbaren (bzw. umschreibbaren) Referenz-Attributgrammatiken von Hedin[4] und Higher-Order Attributgrammatiken[23].

Kastens zeigte in [12], dass es für die sehr mächtige Menge geordneter Attributgrammatiken immer möglich ist eine effiziente Berechnungsstrategie zu bestimmen und einen Evaluator zu generieren, der für alle Eingaben diese Berechnungen durchführt. Die Übersetzerbau-Werkzeugsammlung eli[6] nutzt OAGs zur Generierung vollständiger Übersetzer. Eli unterstützt gleichzeitig bereits eine Reihe von Erweiterungen klassischer Attributgrammatiken wie Vererbung und parametrierbare Module. Für eli wurde ebenfalls der „abstrakte Datentyp zur Namensanalyse“ in [16] vorgestellt.

In [11] werden syntaktische Methoden zur Modularisierung von Attributgrammatiken vorgestellt. Module lassen sich durch Parameter individualisieren und werden instanziiert.

DSLs, die auf dem Erkennen typischer Muster in Attributgrammatiken basieren und Konstruktoren definieren, finden sich in [17]. Weitere Beispiele, wie Verwendung der Definitionstabelle oder Codeerzeugung, werden in [25] vorgestellt.

Erstmals wurde eine Variante des contribution-Musters in [2] vorgestellt. Im Gegensatz zu unserer Variante sind keine in einem Symbol lokalen *via*-Attribute vorgesehen. Auf Boylands Arbeiten aufbauend wird u.a. in [8,4,7,20] eine Variante vorgestellt, die Feld-orientiert arbeitet. Für skalare Datentypen sind Hilfsklassen zu schreiben[20].

Einen Überblick über Attributgrammatiken liefern [3] sowie [22]. Letzteres liefert darüber hinaus einen Überblick zur Verringerung des Aufwands bei der Spezifikation mit Attributgrammatiken.

6 Zusammenfassung

In dieser Arbeit wurden eine Reihe typischer Muster in geordneten Attributgrammtiken vorgestellt. Es wurde gezeigt, wie auf Basis dieser Muster weitere Muster aufgebaut werden können. Die von uns vorgestellten Muster sind kompakter und bieten einen höheren Abstraktionsgrad als die entsprechenden Lösungen, die mit rein Modul-basierten Lösungen wie [11] möglich wären. Für die präsentierten Muster konnten Anwendungsfälle vorgestellt werden. Ebenso wurde auf Speichereinfluss und Laufzeiteinfluss der Muster eingegangen, wenngleich eine genauere Untersuchung und dazugehörige Vorstellung notwendig ist.

Hervorzuheben sind Beiträge als Muster, die in vielen Szenarien eingesetzt werden können und auch als Basis für andere Muster von Nutzen sind. Beiträge, wie sie von Hedin (siehe u.a. [7,4]) verwendet werden, gehen auf die globalen „Collection“-Attribute Boylands (siehe [2]) zurück und verhalten sich ähnlich dem **`constituent`**-Konstrukt aus LIGA und GAG (siehe u.a. [15]) in der Wurzel. Im Gegensatz zu den eben beschriebenen Ansätzen erlauben komplexe Beiträge auch die Einführung oder Verwendung zusätzlicher Attribute ohne direkten Beitrag zu einem Attribut. Damit gehen diese von uns vorgestellten Muster weiter als bisherige Ansätze.

Ebenfalls in [2] erkannt wurden Muster zum Aufsammeln von Attributen, jedoch keine verbesserte Abstraktion als Beiträge dafür gefunden, die bei Boyland noch als Attribute der Wurzel interpretiert werden können. In der bei uns vorliegenden Form bieten sich mehr Einsatzmöglichkeiten.

Die von uns vorgestellte Namensanalyse unterscheidet sich von den in [16] und [5] durch Verwendung des Musters mit Beiträgen und damit einhergehend der Möglichkeit die Definitionstabelle sofort mit semantisch relevanten Informationen zu befüllen. Es existieren weitere Eigenschaften bei der Namensanalyse, wie Namensbereiche, auf die wir bisher nicht eingegangen sind. Eine Herausforderung wird es sein, diese in Muster zu fassen, die mit anderen Mustern kombiniert immernoch eine geordnete Attributgrammatik ergeben.

Es existieren viele Erweiterungen, Bibliotheken und Werkzeuge denen gemein ist, dass sie aufgrund des Erkennens typischer Muster entwickelt wurden, bspw. [11,2,25,16,17]. Dennoch wurden diese typischen Muster nicht genutzt um einen höheren Abstraktionsgrad zu gewinnen, der auf die Semantik des Musters eingeht.

In den (älteren) Übersichtsarbeiten [3] und [22] wird nicht auf typische Muster eingegangen – Ausnahme bildet der Zugriff auf entfernte Attribute. Letzteres ist ähnlich der **Propagation** und dem **Aufsammeln**.

Neben den in Abschnitt 3 beschriebenen Mustern existieren weitere Muster, welche insbesondere auf der Verwendung von Graphen oder Datentypen, basieren. Diese weiteren sowie rein auf Produktionen anwendbaren Muster zu identifizieren und zu analysieren ist das Ziel zukünftiger Forschung. Weiterhin muss ebenfalls noch gezeigt werden, dass die von uns aufgestellten Hypothesen (1 und 2) gültig sind. Wir erwarten von den dazugehörigen Beweisen konkretere Aussagen bzgl. Laufzeiteinfluss und Speichereinfluss als dies in Abschnitt 4 von uns erfolgt ist.

Die von uns gewonnenen Ergebnisse wollen wir in einer DSL zur Verfügung stellen.

Danksagung
Diese Arbeit entstand im Rahmen des vom Bundesministerium für Bildung und Forschung (BMBF) geförderten Forschungsprojekts ELSY (Nr. 16M3202D). Weiterhin danken wir den anonymen Gutachtern früherer Arbeiten, die uns auf die Beiträge in JastAdd aufmerksam gemacht haben.

Literatur

1. Berg, C., Zimmermann, W.: DSL Implementation for Model-based Development of Pumps. In: Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change, pp. 391–406. Springer (2014)
2. Boyland, J.T.: Descriptional Composition of Compiler Components. Ph.D. thesis (1996), aAI9722877
3. Deransart, P., Jourdan, M., Lorho, B.: Attribute Grammars: Definitions, Systems and Bibliography. Springer-Verlag (1988)
4. Ekman, T., Hedin, G.: Rewritable Reference Attributed Grammars. In: Odersky, M. (ed.) ECOOP 2004 – Object-Oriented Programming, Lecture Notes in Computer Science, vol. 3086, pp. 147–171. Springer (2004)
5. Ekman, T., Hedin, G.: Modular Name Analysis for Java Using JastAdd. In: Lämmel, R., Saraiva, J.a., Visser, J. (eds.) Generative and Transformational Techniques in Software Engineering, Lecture Notes in Computer Science, vol. 4143, pp. 422–436. Springer Berlin Heidelberg (2006), http://dx.doi.org/10.1007/11877028_18
6. Gray, R.W., Levi, S.P., Heuring, V.P., Sloane, A.M., Waite, W.M.: Eli: A Complete, Flexible Compiler Construction System. Communications of the ACM 35(2), 121–130 (1992)
7. Hedin, G.: An Introductory Tutorial on JastAdd Attribute Grammars. In: Fernandes, J.a., Lämmel, R., Visser, J., Saraiva, J.a. (eds.) Generative and Transformational Techniques in Software Engineering III, Lecture Notes in Computer Science, vol. 6491, pp. 166–200. Springer Berlin Heidelberg (2011), http://dx.doi.org/10.1007/978-3-642-18023-1_4
8. Hedin, G.: Reference Attributed Grammars. Informatica (Slovenia) 24(3), 301–317 (2000)
9. Johnson, R., Helm, R., Vlissides, J., Gamma, E.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional (1995)
10. Kadhim, B., Waite, W.: Maptool — supporting modular syntax development. In: Gyimóthy, T. (ed.) Compiler Construction, Lecture Notes in Computer Science, vol. 1060, pp. 268–280. Springer-Verlag (1996), http://dx.doi.org/10.1007/3-540-61053-7_67
11. Kastens, U., Waite, W.: Modularity and reusability in attribute grammars. Acta Informatica 31(7), 601–627 (1994), http://dx.doi.org/10.1007/BF01177548

12. Kastens, U.: Ordered Attributed Grammars. Acta Informatica 13(3), 229–256 (1980)
13. Kastens, U.: Lifetime analysis for attributes. Acta Informatica 24(6), 633–652 (1987)
14. Kastens, U.: Attribute Grammars as a specification method. In: Alblas, H., Melichar, B. (eds.) Attribute Grammars, Applications and Systems, Lecture Notes in Computer Science, vol. 545, pp. 16–47. Springer Berlin Heidelberg (1991), `http://dx.doi.org/10.1007/3-540-54572-7_2`
15. Kastens, U., Hutt, B., Zimmermann, E.: GAG, a practical compiler generator, Lecture Notes in Computer Science, vol. 141. Springer-Verlag (1982)
16. Kastens, U., Waite, W.M.: An abstract data type for name analysis. Acta Informatica 28(6), 539–558 (1991)
17. Kastens, U., Waite, W.M.: Reusable specification modules for type analysis. Software: Practice and Experience 39(9), 833–864 (2009), `http://dx.doi.org/10.1002/spe.917`
18. Knuth, D.E.: Semantics of Context-Free Languages. Mathematical systems theory 2(2), 127–145 (1968)
19. Koskimies, K.: Object-orientation in attribute grammars. In: Attribute Grammars, Applications and Systems. pp. 297–329. Springer-Verlag (1991)
20. Magnusson, E., Ekman, T., Hedin, G.: Extending attribute grammars with collection attributes–evaluation and applications. In: Source Code Analysis and Manipulation, 2007. SCAM 2007. Seventh IEEE International Working Conference on. pp. 69–80 (Sept 2007)
21. Mernik, M., Heering, J., Sloane, A.M.: When and How to Develop Domain-Specific Languages. ACM Comput. Surv. 37(4), 316–344 (Dec 2005)
22. Paakki, J.: Attribute grammar paradigms – a high-level methodology in language implementation. ACM Comput. Surv. 27(2), 196–255 (Jun 1995), `http://doi.acm.org/10.1145/210376.197409`
23. Vogt, H.H., Swierstra, S.D., Kuiper, M.F.: Higher Order Attribute Grammars. In: Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation. pp. 131–145. PLDI '89, ACM (1989)
24. Voigtländer, J.: Bidirectionalization for free!(Pearl). ACM SIGPLAN Notices 44(1), 165–176 (2009)
25. Waite, W., Kastens, U., Sloane, A.M.: Generating Software from Specifications. Jones and Bartlett Publishers, Inc., USA (2007)

A New Foundation for Computing Science

Are we Studying the Right Things ?

Dines Bjørner[*]
Fredsvej 11, DK-2840 Holte, Denmark
DTU, DK-2800 Kgs. Lyngby, Denmark
E–Mail: bjorner@gmail.com, URL: www.imm.dtu.dk/~dibj

September 2, 2015

Abstract

We argue that computing systems requirements must be based on precisely described domain models. We further argue that domain science & engineering offers a new dimension in computing. We review our work in this area and we hint at a research and experimental engineering programme for the first two phases of the triptych of domain enginering, requirements engineering and software design.

1 Introduction

This author can refer to some substantial evidence [19, 21, 33] that using formal specifications in software development brings some substantial benefits. Section 2 recalls a first, 1981–1984, instance of such benefits. Yet, as also outlined in [15, *Bjørner & Havelund: 40 Years of Formal Methods — 10 Obstacles and 3 Possibilities*], "propagation" of formal methods into a wider industry seems lacking. Although [35, Woodcock et al.] lacks a reference to the formal methods project covered in Sect. 2, it is a fair reference to a number of projects supporting the author's "benefits" claim.

1.1 The Domain Engineering Claim

In this paper we wish, however, to not "push" the *formal methods* claim, but to "push" a, or the, *domain science & engineering* claim: *in order to* **design software** *one must have a good grasp of its requirements; in order to* **prescribe requirements** *one must have a good grasp of the underlying domain; so we expect that behind every serious software development there lies a stable* **domain description**. This, then, is the purpose of this paper: to "tout" the concept of domain science and engineering, emphasizing, in this paper, the latter.

1.2 Aim of Paper

So this is neither a *theory* nor a *programming methodology* paper. It is a review paper: *"where do we stand ?"* with respect to being able to develop correct software and software that meets customers' expectations ?; and *"how can those two issues: 'correctness' and 'meeting expectations' be improved ?"*.

[*] This paper is the background paper for a 15 minute presentation to be given at KPS 2015, Parkhotel Pörtschach, Wörthersee, Austria, October 5–7, 2015. The presentation is, obviously, expected to be approximately 12–15 slides !

1.3 Structure of Paper

Section 2 brings two examples: one of arbitrary, but well-formed transportation nets (illustrated by a road net), the other of arbitrary, but well-formed pipelines with the flow (laws) of liquid materials. The purpose of Sect. 2 is to review a 44 man-year project using formal methods ("lightly"). We bring this example — of a now more than 30 year old project (1981–1984) — to show an early use of a carefully narrated formal domain description, a project that we claim to have been a very successful one. Section 3 overviews our concept of *TripTych* development: from domain descriptions, via requirements prescriptions, to software design. We emphasize the domain science & engineering aspects. Section 4 Discusses our claim that this *TripTych* suggests a new foundation for computing science.

2 A Background Development

We sketch the structure of a successful 44 man year project which developed a commercial compiler according to the *TripTych* approach and using formal specifications — with success measured in therms of meeting customers' expectations and being correct.

2.1 The 1981–1984 DDC Ada Compiler Development Project

In the spring semester (6 months) of 1980 five MSc students worked out their MSc theses: **A Formal Description of Ada**. The four theses were published as [19]. That work became the basis for a full-scale industry-size project: *The DDC*[1] *ADA Compiler Project*, funded, in part by the *CEC*, the *Commission of the European Countries*. The project was carried out according to abstraction and refinement principles — as far as the ⋯ dotted box: the leftmost dynamic semantics (quadruple of) boxes[2] as well as the *A-code Language* and *Compiling Algorithm* is concerned — laid down in [2], and can be diagrammed as shown in Fig. 1 We explain the approach taken to develop, using formal specifications, an industry-strength, commercial compiler for the *US DoD*[3] *Ada* programming language. We do so using Fig. 1 as a reference point. Each box represents a specification and denotes a mathematical object. Each directed line between boxes represents a step of development, from a higher to a lower level of abstraction, and denotes a proof (of correctness, also a mathematical object). There were three phases of development: the *domain* engineering phase, the *requirements engineering* phase, and the *software design* phase. They are clearly marked in Fig. 1. First a formal description was developed for Ada. This phase is referred to as the 'Domain'. It had four stages: first the *Abstract Syntax*, then (developed "concurrently") the *Higher-order Static Semantics*, the *"Denotational" Dynamic Sequential Semantics* and the *"Operational" Dynamic Parallel Semantics*. Then a phase, *Requirements*, consisting of several stages. The refinement work represented by each of the boxes, were conditioned by various requirements. But we show such only for two boxes: dashed, labelled pointed lines. The *Higher-order Static Semantics* is refined in two stages: first a *Resumption Static Semantics* and then a *First-order Static Semantics*. The *"Denotational" Dynamic Sequential Semantics* was, in principle, refined in three stages: a *1st-order Functional Interpreter*, a *Imperative Stack* dynamic semantics and a *Macro-expansion* dynamic semantics. From the *Operational "Parallel" Semantics* was developed an operational *Run-time Semantics* for the concurrency constructs of Ada. From the *Macro-expansion* semantics was developed the design of an *A*[da] *Code Language* which was given a semantics commensurate with the specification language and *Macro-expansion* semantics. And from the *Macro-expansion* semantics and the *A Code Language* was developed a *Compiling Algorithm* which to every construct of Ada prescribed a sequence of *A Code*. The *Run-time Interpreter* was developed from the *Operational "Parallel" Ada Semantics*. Two *requirements assumptions* were: the compiler should execute within a 128 KB addressing space, and the compiled code should likewise execute within a 128 KB addressing space. Therefore the compiler need be decomposed into a number of passes where a pass was defined as that of a linear

[1] DDC: Dansk Datamatik Center was an industry-operated R&D centre, 1979–1989.

[2] *"Denotational" Sequental Ada, 1st-order Functional Interpreter, Imperative Stack* and *Macro-Expansion*

[3] DoD: Department of Defense

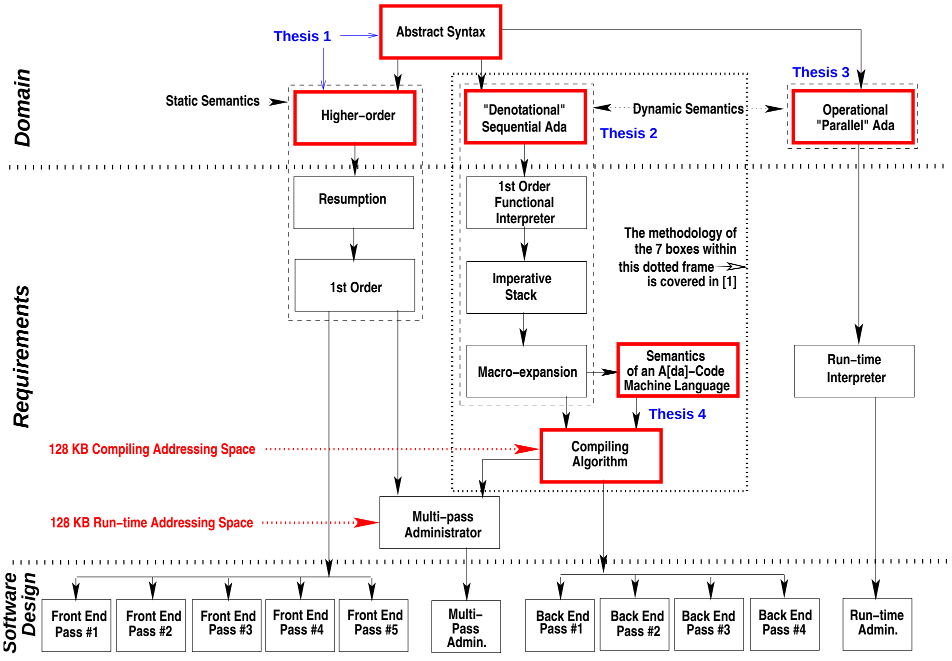


Figure 1: The Ada Compiler Software Development Graph. Bold-faced Boxes published in [19]

reading of of the Ada program text either left-to-right (forwards), or right-to-left (backwards), and in either pre-, in- or post-order[4]. From the combined *1st-order Static Semantics* and the *Compiling Algorithm* was, after careful analysis of these, developed a specification for a multi-pass administrator. The multi-pass analysis and synthesis resulted in five passes for the statics checks (i.e., "front-end"), and four passes for the code generator (i.e., "back-end"). These concluded the domain and requirements phases which were all specified in VDM [16, 31] for a total of approximately 10.000, respectively 56,000 lines of VDM and formula annotations. The nine compiler *Passes*, *Multi-pass Administrator*, and the *Run-time Administrator* were all coded from their specifications in a subset of the Ada language for which a compiler was developed in parallel with the full-Ada development!

2.2 A Review

2.2.1 Resources

The above project took place more than 30 years ago! Approximately the following man-power resources were used: For the *Domain* phase: seven people, one year; for the *Requirements* phase (exclusive of the *Multi-pass Administrator*: eleven people, one year; for the *Multi-pass Administrator*: six people, half a year; and for the rest (nine *Passes* and the *Administrators*): 12 people, 14 months. The subset Ada compiler development consumed seven man years. Thus a total of 42 man years was spent on effective development and its management, 2 man years on management of donors, funding and marketing.

[4] Pre-order: visiting program phrase tree nodes when first encountered; in-order: any time encountered, or post-order: when last encountered.

2.2.2 Formal Methods "Lite"

VDM was the prime "carrier" of the Ada compiler development. The domain and the requirements phases were specified in VDM. No properties of these specifications were formalised let alone proved. The first 10 years of use by industry on three continents (China, Japan, USA and Europe) revealed few, and only trivial errors: less than 2% of original development resources were spent on error corrections with average "repair" times being in the order of 1–2 days.

2.2.3 Epilogue

The above-outlined Ada compiler development project was reported in [21, 33]. The use of formal methods was clear. But 'formal methods' were not used in any other sense than formal specifications. Properties of and relationships between stage specifications, i.e., boxes, were not formalised. And yet, the project must be judged an unqualified success for formal methods. It took far fewer manpower resources than any other Ada compiler development project in those days. It had far, far fewer "bugs" than any comparable software development project in those days or since. Yet there were no tools available: No VDM syntax checker, No specification analyser. No nothing!

3 The Triptych of Software Engineering

We suggest a `TripTych` view of software engineering: *before software can be designed and coded we must have a reasonable grasp of "its" requirements; and before requirements can be prescribed we must have a reasonable grasp of "the underlying" domain.* To us, therefore, software engineering contains the three sub-disciplines:

- domain engineering,
- requirements engineering and
- software design.

3.1 What's New ?

So *"What's New ?"* in this ? Well, as far as the surveyed compiler development is concerned, nothing: that is how one should develop compilers — although it seems that it was done only once![5] What can we learn from the example of Sect. 2 ? We can postulate that when there is a formal understanding of the domain — and of the stages from domain to requirements and on to software design, then software can be developed with greater assurance of meeting users' expectations and be correct than if not! So that is what we are therefore proposing: to treat the domain, the application area for software development, as "a language" whose terms designate phenomena in the domain and "spoken/uttered" about by practitioners in the domain. So we consider a domain description to be the description of the syntax and the semantics of a language.

3.2 Domain Science & Engineering

3.2.1 What is a Domain ?

A **domain** is a human- and artifact-assisted arrangement of *endurant*, that is spatially "stable", and *perdurant*, that is temporally "fleeting" entities.

[5] Most textbooks in compiler development do not cover neither static nor dynamic semantics formally — and they certainly do not motivate the run-time stack stack/unstack operations upon procedure calls and returns such as done in [2].

3.2.2 Example Domains

To help understand the above delineation of the 'domain' concept we list some examples for which we can also refer to some either published or reported domain descriptions:

Example 1 . Manifest Domain Names: Examples of suggestive names of manifest domains are: *air traffic*, *banks*, *container lines*, *documents*, *hospitals*, *manufacturing*, *pipelines*, *railways* and *road nets*.

3.2.3 Comparison to Other Sciences and Their Engineering

We focus on the natural sciences and their engineerings: civil (or construction) engineering (buildings, roads, bridges, tunnels, etc.), mechanical engineering, chemical engineering, electrical (power engineering), electronics engineering (VLSI, IT hardware, etc.) and radio engineering (radio waves, transmitters, receivers, etc.). For all of these related technologies engineers are properly educated, knows the underlying sciences, that is, the domains of their artifacts. Not so, today, 2015, for software engineers for the domains listed in Example 1. Software engineers asked to develop software for either of air traffic control, banking. container lines, health care, railways, road pricing, etcetera, are expected to find out, themselves, what the relevant domain is, how it behaves, etc. No wonder that it often fails!

3.2.4 Domain Descriptions: Internet References

Now, we would not postulate the above without firm evidence. *"Proof in the pudding"* sort-of-evidence that domains can indeed be properly, informally and formally described. We shall first mention some existing descriptions before we exemplify fragments of such descriptions.

We list a number of reports all of which document descriptions of domains. These descriptions were carried out, by the present author, in order to research and develop the domain analysis and description concepts now summarised in the present paper. These reports ought now be revised, some slightly, others less so, so as to follow all of the prescriptions of the current paper. Except where a URL is given in full, please prefix the web reference with: `http://www2.compute.dtu.dk/~dibj/`.

1 *A Railway Systems Domain* D.Bjørner et al.

- Scheduling and Rescheduling of Trains; C.W.George and S.Prehn, 1996, `amore/scheduling.pdf`
- Formal Software Techniques in Railway Systems; 2000, `amore/dines-fac.pdf`
- Dynamics of Railway Systems; 2000, `amore/ifac-dynamics.pdf`
- Railway Staff Rostering; A.Strupchanska et al., 2003, `amore/albena-amore.pdf`
- Train Maintenance Routing; M.Peñicka et al., 2003, `amore/martin-amore.pdf`
- Train Composition and Decomposition: Domain and Requirements (draft), P.Karras et al., 2003, `amore/panos-amore.pdf`

2 *Models of IT Security. Security Rules & Regulations*, `it-security.pdf`, 2006. See [13]. A sketch is given of the IT security rules laid down by ISO

3 *A Container Line Industry Domain*, `container-paper.pdf`, 2007

4 *The "Market": Consumers, Retailers, Wholesalers, Producers*, `themarket.pdf`, 2007 See [3].

5 *What is Logistics ?* `logistics.pdf`, 2009

6 *A Domain Model of Oil Pipelines*, `pipeline.pdf`, 2009

7 *Transport Systems*, `comet/comet1.pdf`, 2010

8 *The Tokyo Stock Exchange*, `todai/tse-1.pdf` and `todai/tse-2.pdf`, 2010

9 *On Development of Web-based Software. A Divertimento*, `wfdftp.pdf`, 2010

10 *Documents (incomplete draft)*, `doc-p.pdf`, See [12]. 2013

3.2.5 An Example: Road Nets, Vehicles and Traffic

Parts The root domain, $\Delta_{\mathcal{D}}$, whose description is to be exemplified, is that of a composite traffic system (1a.) with a road net, (1b.) with a fleet of vehicles and (1c.) of whose individual position on the road net we can speak, that is, monitor.

1 We analyse the traffic system into
   a a composite road net,
   b a composite fleet (of vehicles), and
   c an atomic monitor.

2 The road net consists of two composite parts,
   a an aggregation of hubs and
   b an aggregation of links.

type
1. $\Delta_\triangle$
1a. N$_\triangle$
1b. F$_\triangle$
1c. M$_\triangle$
value
1a. **obs_part_N**$_\triangle$: $\Delta_\triangle \to$ N$_\triangle$
1b. **obs_part_F**$_\triangle$: $\Delta_\triangle \to$ F$_\triangle$
1c. **obs_part_M**$_\triangle$: $\Delta_\triangle \to$ M$_\triangle$
type
2a. HA$_\triangle$
2b. LA$_\triangle$
value
2a. **obs_part_HA**$_\triangle$: N$_\triangle \to$ HA$_\triangle$
2b. **obs_part_LA**$_\triangle$: N$_\triangle \to$ LA$_\triangle$ ■

3 Hub aggregates are sets of hubs.

4 Link aggregates are sets of links.

5 Fleets are sets of vehicles.

6 We introduce some auxiliary functions.
   a links extracts the links of a network.
   b hubs extracts the hubs of a network.

type
3. H$_\triangle$, HS$_\triangle$ = H$_\triangle$-**set**
4. L$_\triangle$, LS$_\triangle$ = L$_\triangle$-**set**
5. V$_\triangle$, VS$_\triangle$ = V$_\triangle$-**set**
value
3. **obs_part_HS**$_\triangle$: HA$_\triangle \to$ HS$_\triangle$
4. **obs_part_LS**$_\triangle$: LA$_\triangle \to$ LS$_\triangle$
5. **obs_part_VS**$_\triangle$: F$_\triangle \to$ VS$_\triangle$
6a. links$_\triangle$: $\Delta_\triangle \to$ L-**set**
6a. links$_\triangle$($\delta_\triangle$) $\equiv$ **obs_part_LS**(**obs_part_LA**($\delta_\triangle$))
6b. hubs$_\triangle$: $\Delta_\triangle \to$ H-**set**
6b. hubs$_\triangle$($\delta_\triangle$) $\equiv$ **obs_part_HS**(**obs_part_HA**($\delta_\triangle$)) ■

Unique Identifiers We cover the unique identifiers of all parts, whether needed or not.

7 Nets, hub and link aggregates, hubs and links, fleets, vehicles and the monitor all
   a have unique identifiers
   b such that all such are distinct, and
   c with corresponding observers.

8 We introduce some auxiliary functions:
   a xtr_lis extracts all link identifiers of a traffic system.
   b xtr_his extracts all hub identifiers of a traffic system.
   c given an appropriate link identifier and a net get_link 'retrieves' the designated link.
   d given an appropriate hub identifier and a net get_hub 'retrieves' the designated hub.

type
7a. NI, HAI, LAI, HI, LI, FI, VI, MI
value
7c. **uid_NI**: N$_\triangle \to$ NI
7c. **uid_HAI**: HA$_\triangle \to$ HAI
7c. **uid_LAI**: LA$_\triangle \to$ LAI
7c. **uid_HI**: H$_\triangle \to$ HI
7c. **uid_LI**: L$_\triangle \to$ LI
7c. **uid_FI**: F$_\triangle \to$ FI
7c. **uid_VI**: V$_\triangle \to$ VI
7c. **uid_MI**: M$_\triangle \to$ MI
axiom
7b. NI$\bigcap$HAI=$\emptyset$, NI$\bigcap$LAI=$\emptyset$, NI$\bigcap$HI=$\emptyset$, etc.

where axiom 7b is expressed semi-formally, in mathematics.

```
value
8a.  xtr_lis: $\Delta_\triangle$ → LI-set
8a.  xtr_lis($\delta_\triangle$) ≡
8a.     let ls = links($\delta_\triangle$) in {uid_LI(l)|l:L•l ∈ ls} end
8b.  xtr_his: $\Delta_\triangle$ → HI-set
8b.  xtr_his($\delta_\triangle$) ≡
8b.     let hs = hubs($\delta_\triangle$) in {uid_HI(h)|h:H•k ∈ hs} end
8c.  get_link: LI → $\Delta_\triangle$ $\overset{\sim}{\rightarrow}$ L
8c.  get_link(li)($\delta_\triangle$) ≡
8c.     let ls = links($\delta_\triangle$) in
8c.     let l:L • l ∈ ls ∧ li=uid_LI(l) in l end end
8c.     pre: li ∈ xtr_lis($\delta_\triangle$)
8d.  get_hub: HI → $\Delta_\triangle$ $\overset{\sim}{\rightarrow}$ H
8d.  get_hub(hi)($\delta_\triangle$) ≡
8d.     let hs = hubs($\delta_\triangle$) in
8d.     let h:H • h ∈ hs ∧ hi=uid_HI(h) in h end end
8d.     pre: hi ∈ xtr_his($\delta_\triangle$)   ■
```

Mereology

9 Links are connected to exactly two distinct hubs.

10 Hubs are connected to zero or more links.

11 For a given net the link and hub identifiers of the mereology of hubs and links must be those of links and hubs, respectively, of the net.

```
type
9.   LM′ = HI-set, LM = {|his:HI-set • card(his)=2|}
10.   HM = LI-set
value
9.   mereo_L: L → LM
10.   mereo_H: H → HM
axiom [Well−formedness of Road Nets, N]
11.   ∀ n:N,l:L,h:H•  l ∈ obs_part_Ls(obs_part_LC(n))∧h ∈ obs_part_Hs(obs_part_GC(n))
11.     let his=mereology_H(l), lis=mereology_H(h) in
11.     his⊆∪{uid_H(h) | h ∈ obs_part_Hs(obs_part_HC(n))}
11.    ∧ lis⊆∪{uid_H(l) | l ∈ obs_part_Ls(obs_part_LC(n))} end
```

Attributes We may not have shown all of the attributes mentioned below — so consider them informally introduced!

- **Hubs:** *location*s[6] are considered static, *wear and tear* (condition of road surface) is considered inert, *hub state*s and *hub state space*s are considered programmable;
- **Links:** *length*s and *location*s are considered static, *wear and tear* (condition of road surface) is considered inert, *link state*s and *link state space*s are considered programmable;

[6] By location we mean a cadestral/geodetic position.

- **Vehicles:** *manufacturer name*, *engine type* (whether diesel, gasoline or electric) and *engine power* (kW/horse power) are considered static; *velocity* and *acceleration* may be considered reactive (i.e., a function of gas pedal position, etc.), *global position* (informed via a `GNSS: Global Navigation Satellite System`) and *local position* (calculated from a global position) are considered biddable ■

We treat one attribute each for hubs, links, vehicles and the monitor. First we treat hubs.

12 Hubs

  a have *hub states* which are sets of pairs of identifiers of links connected to the hub[7],

  b and have *hub state spaces* which are sets of hub states[8].

13 For every net,

  a link identifiers of a hub state must designate links of that net.

  b Every hub state of a net must be in the hub state space of that hub.

14 Hubs have geodetic and cadestral location.

15 We introduce an auxiliary function: xtr_lis extracts all link identifiers of a hub state.

```
type
12a.  HΣ = (LI×LI)-set
12b.  HΩ = HΣ-set
value
12a.  attr_HΣ: H → HΣ
12b.  attr_HΩ: H → HΩ
axiom
13.   ∀ δ:Δ,
13.     let hs = hubs(δ) in
13.     ∀ h:H • h ∈ hs •
13a.        xtr_lis(h)⊆xtr_lis(δ)
13b.        ∧ attr_Σ(h) ∈ attr_Ω(h)
13.     end
type
14.   HGCL
value
14.   attr_HGCL: H → HGCL
15.   xtr_lis: H → LI-set
15.   xtr_lis(h) ≡
15.     {li | li:LI,(li′,li″):LI×LI •
15.         (li′,li″) ∈ attr_HΣ(h) ∧ li ∈ {li′,li″}}
```

Then links.

16 Links have lengths.

17 Links have geodetic and cadestral location.

18 Links have states and state spaces:

  a States modeled here as pairs, (hi', hi''), of identifiers the hubs with which the links are connected and indicating directions (from hub h' to hub h''.) A link state can thus have 0, 1, 2, 3 or 4 such pairs.

  b State spaces are the set of all the link states that a link may enjoy.

```
type
16.   LEN
17.   LGCL
18a.  LΣ = (HI×HI)-set
18b.  LΩ = LΣ-set
value
16.   attr_LEN: L → LEN
17.   attr_LGCL: L → LGCL
18a.  attr_LΣ: L → LΣ
18b.  attr_LΩ: L → LΩ
axiom
18.   ∀ n:N •
18.     let ls = xtr−links(n), hs = xtr_hubs(n) in
18.     ∀ l:L•l ∈ ls ⇒
```

[7] A hub state “signals” which input-to-output link connections are open for traffic.

[8] A hub state space indicates which hub states a hub may attain over time.

```
18a.      let lσ = attr_LΣ(l) in
18a.      0≤card lσ≤4
18a.    ∧ ∀ (hi′,hi″):(HI×HI)•(hi′,hi″) ∈ lσ ⇒
18a.             {get_H(hi′)(n),get_H(hi″)(n)}=mereo_L(l)
18b.      ∧ attr_LΣ(l) ∈ attr_LΩ(l) ■
18.     end end
```

Then vehicles.

19 Every vehicle of a traffic system has a position which is either ‘on a link’ or ‘at a hub’.

a An ‘on a link’ position has four elements: a unique link identifier which must designate a link of that traffic system and a pair of unique hub identifiers which must be those of the mereology of that link.

b The ‘on a link’ position real is the fraction, thus properly between 0 (zero) and 1 (one) of the length from the first identified hub “down the link” to the second identifier hub.

c An ‘at a hub’ position has three elements: a unique hub identifier and a pair of unique link identifiers — which must be in the hub state.

```
type
19.    VPos = onL | atH
19a.   onL :: LI HI HI R
19b.   R = Real        axiom ∀ r:R • 0≤r≤1
19c.   atH :: HI LI LI
value
19.   attr_VPos: V_Δ → VPos
axiom
19a.   ∀ n_Δ:N_Δ, onL(li,fhi,thi,r):VPos •
19a.       ∃ l_Δ:L_Δ•l_Δ∈obs_part_LS(obs_part_N_Δ(n_Δ))
19a.           ⇒ li=uid_L_Δ(l)∧{fhi,thi}=mereo_L_Δ(l_Δ),
19c.   ∀ n_Δ:N_Δ, atH(hi,fli,tli):VPos •
19c.       ∃ h_Δ:H_Δ•h_Δ∈obs_part_HS_Δ(obs_part_N(n_Δ))
19c.           ⇒ hi=uid_H_Δ(h_Δ)∧(fli,tli) ∈ attr_LΣ(h_Δ)
```

And finally monitors. We consider only one monitor attribute.

20 The monitor has a vehicle traffic attribute.

a For every vehicle of the road transport system the vehicle traffic attribute records a possibly empty list of time marked vehicle positions.

b These vehicle positions are alternate sequences of ‘on link’ and ‘at hub’ positions

i such that any sub-sequence of ‘on link’ positions record the same link identifier, the same pair of ‘’to’ and ‘from’ hub identifiers and increasing fractions,

ii such that any sub-segment of ‘at hub’ positions are identical,

iii such that vehicle transition from a link to a hub is commensurate with the link and hub mereologies, and

iv such that vehicle transition from a hub to a link is commensurate with the hub and link mereologies.

```
type
20.  Traffic = VI →_m (T × VPos)*
value
20.  attr_Traffic: M → Traffic
axiom
20b.  ∀ δ:Δ •
20b.      let m = obs_part_M_Δ(δ) in
20b.      let tf = attr_Traffic(m) in
20b.      dom tf ⊆ xtr_vis(δ) ∧
```

```
20b.        ∀ vi:VI • vi ∈ dom tf •
20b.            let tr = tf(vi) in
20b.            ∀ i,i+1:Nat • {i,i+1}⊆dom tr •
20b.                let (t,vp)=tr(i),(t′,vp′)=tr(i+1) in
20b.                t<t′
20(b)i.              ∧ case (vp,vp′) of
20(b)i.                    (onL(li,fhi,thi,r),onL(li′,fhi′,thi′,r′))
20(b)i.                       → li=li′∧fhi=fhi′∧thi=thi′∧r≤r′
20(b)i.                        ∧ li ∈ xtr_lis(δ)
20(b)i.                        ∧ {fhi,thi} = mereo_L(get_link(li)(δ)),
20(b)ii.                   (atH(hi,fli,tli),atH(hi′,fli′,tli′))
20(b)ii.                      → hi=hi′∧fli=fli′∧tli=tli′
20(b)ii.                       ∧ hi ∈ xtr_his(δ)
20(b)ii.                       ∧ (fli,tli) ∈ mereo_H(get_hub(hi)(δ)),
20(b)iii.                  (onL(li,fhi,thi,1),atH(hi,fli,tli))
20(b)iii.                     → li=fli∧thi=hi
20(b)iii.                      ∧ {li,tli} ⊆ xtr_lis(δ)
20(b)iii.                      ∧ {fhi,thi}=mereo_L(get_link(li)(δ))
20(b)iii.                      ∧ hi ∈ xtr_his(δ)
20(b)iii.                      ∧ (fli,tli) ∈ mereo_H(get_hub(hi)(δ)),
20(b)iv.                   (atH(hi,fli,tli),onL(li′,fhi′,thi′,0))
20(b)iv.                      →  etcetera,
20b.                   _  → false
20b.        end end end end end  ■
```

3.2.6 Another Example: Pipelines

Parts

21 A pipeline consists of an indefinite number of pipeline units.

22 A pipeline units is either a well, or a pipe, or a pump, or a valve, or a fork, or a join, or a sink.

23 All these unit sorts are atomic and disjoint.

```
type
21.   PL, U, We, Pi, Pu, Va, Fo, Jo, Si
21.   Well, Pipe, Pump, Valv, Fork, Join, Sink
value
21.   obs_part_Us: PL → U-set
type
22.   U == We | Pi | Pu | Va | Fo | Jo | Si
23.  We::Well, Pi::Pipe, Pu::Pump, Va::Valv, Fo:Fork, Jo::Join, Si::Sink
```

Unique Identifiers

24 Every pipeline unit has a unique identifier.

```
type
24.  UI
value
24.  uid_U: U → UI
```

Materials

25 Applying `obs_material_sorts_U` to any pipeline unit, u:U, yields

- a a type clause stating the material sort LoG for some further undefined liquid or gaseous material, and b a material observer function signature.

```
     type
25a      LoG
     value
25b      obs_mat_LoG: U → LoG
```

Mereology Pipeline units serve to conduct fluid or gaseous material. The flow of these occur in only one direction: from so-called input to so-called output.

26 Wells have exactly one connection to an output unit.

27 Pipes, pumps and valves have exactly one connection from an input unit and one connection to an output unit.

28 Forks have exactly one connection from an input unit and exactly two connections to distinct output units.

29 Joins have exactly one two connection from distinct input units and one connection to an output unit.

30 Sinks have exactly one connection from an input unit.

31 Thus we model the mereology of a pipeline unit as a pair of disjoint sets of unique pipeline unit identifiers.

```
type
31.  UM′=(UI-set×UI-set)
31.  UM={|(iuis,ouis):UI-set×UI-set•iuis ∩ ouis={}|}
value
31.  mereo_U: UM
axiom [Well−formedness of Pipeline Systems, PLS (0)]
     ∀ pl:PL,u:U • u ∈ obs_part_Us(pl) ⇒
             let (iuis,ouis)=mereo_U(u) in
             case (card iuis,card ouis) of
26.             (0,1) → is_We(u),
27.             (1,1) → is_Pi(u)∨is_Pu(u)∨is_Va(u),
28.             (1,2) → is_Fo(u),
29.             (2,1) → is_Jo(u),
30.             (1,0) → is_Si(u)
             end end
```

Attributes Let us postulate a[n attribute] sort Flow. We now wish to examine the flow of liquid (or gaseous) material in pipeline units. We use two types

32 F for "productive" flow, and L for wasteful leak.

Flow and leak is measured, for example, in terms of volume of material per second. We then postulate the following unit attributes "measured" at the point of in- or out-flow or in the interior of a unit.

33 current flow of material into a unit input connector,

34 maximum flow of material into a unit input connector while maintaining laminar flow,

35 current flow of material out of a unit output connector,

36 maximum flow of material out of a unit output connector while maintaining laminar flow,

37 current leak of material at a unit input connector,

38 maximum guaranteed leak of material at a unit input connector,

39 current leak of material at a unit input connector,

40 maximum guaranteed leak of material at a unit input connector,

41 current leak of material from "within" a unit, and

42 maximum guaranteed leak of material from "within" a unit.

```
type
32.  F, L
value
33.  attr_cur_iF: U → UI → F
34.  attr_max_iF: U → UI → F
35.  attr_cur_oF: U → UI → F
36.  attr_max_oF: U → UI → F
37.  attr_cur_iL: U → UI → L
38.  attr_max_iL: U → UI → L
39.  attr_cur_oL: U → UI → L
40.  attr_max_oL: U → UI → L
41.  attr_cur_L: U → L
42.  attr_max_L: U → L
```

The maximum flow attributes are static attributes and are typically provided by the manufacturer as indicators of flows below which laminar flow can be expected. The current flow attributes are dynamic attributes

Intra Unit Flow and Leak Law

43 For every unit of a pipeline system, except the well and the sink units, the following law apply.

44 The flows into a unit equal

a the leak at the inputs

b plus the leak within the unit

c plus the flows out of the unit

d plus the leaks at the outputs.

```
axiom [Well−formedness of Pipeline Systems, PLS (1)]
43.   ∀ pls:PLS,b:B\We\Si,u:U •
43.       b ∈ obs_part_Bs(pls)∧u=obs_part_U(b)⇒
43.       let (iuis,ouis) = mereo_U(u) in
44.       sum_cur_iF(iuis)(u) =
44a.          sum_cur_iL(iuis)(u)
44b.        ⊕ attr_cur_L(u)
44c.        ⊕ sum_cur_oF(ouis)(u)
44d.        ⊕ sum_cur_oL(ouis)(u)
43.       end
```

45 The sum_cur_iF (cf. Item 44) sums current input flows over all input connectors.

46 The sum_cur_iL (cf. Item 44a) sums current input leaks over all input connectors.

47 The sum_cur_oF (cf. Item 44c) sums current output flows over all output connectors.

48 The sum_cur_oL (cf. Item 44d) sums current output leaks over all output connectors.

```
45.   sum_cur_iF: UI-set → U → F
45.   sum_cur_iF(iuis)(u) ≡ ⊕ {attr_cur_iF(ui)(u)|ui:UI•ui ∈ iuis}
46.   sum_cur_iL: UI-set → U → L
46.   sum_cur_iL(iuis)(u) ≡ ⊕ {attr_cur_iL(ui)(u)|ui:UI•ui ∈ iuis}
47.   sum_cur_oF: UI-set → U → F
47.   sum_cur_oF(ouis)(u) ≡ ⊕ {attr_cur_iF(ui)(u)|ui:UI•ui ∈ ouis}
48.   sum_cur_oL: UI-set → U → L
48.   sum_cur_oL(ouis)(u) ≡ ⊕ {attr_cur_iL(ui)(u)|ui:UI•ui ∈ ouis}
      ⊕: (F|L) × (F|L) → F
```

where ⊕ is both an infix and a distributed-fix function which adds flows and or leaks

Inter Unit Flow and Leak Law

49 For every pair of connected units of a pipeline system the following law apply:

a the flow out of a unit directed at another unit minus the leak at that output connector

b equals the flow into that other unit at the connector from the given unit plus the leak at that connector.

```
    axiom [Well−formedness of Pipeline Systems, PLS (2)]
49.     ∀ pls:PLS,b,b′:B,u,u′:U•
49.         {b,b′}⊆obs_part_Bs(pls)∧b≠b′∧u′=obs_part_U(b′)
49.          ∧ let (iuis,ouis)=mereo_U(u),(iuis′,ouis′)=mereo_U(u′),
49.                ui=uid_U(u),ui′=uid_U(u′) in
49.            ui ∈ iuis ∧ ui′ ∈ ouis′ ⇒
49a.              attr_cur_oF(u′)(ui′) − attr_leak_oF(u′)(ui′)
49b.           = attr_cur_iF(u)(ui) + attr_leak_iF(u)(ui)
49.            end
49.     comment: b′ precedes b
```

From the above two laws one can prove the **theorem:** what is pumped from the wells equals what is leaked from the systems plus what is output to the sinks.

3.2.7 Domain Descriptions: Methodology

By a **method** we shall understand a set of **principles** for **selecting** and **applying** **techniques** and **tools** for **constructing artifacts** By **methodology** we shall understand the **study** and **knowledge** of **methods**.

The tools of the domain description method centers around two kinda of **prompts**. By a **prompt** we shall understand something that induces an action, an occasion or incitement to inspire, or an assist suggesting something to be expressed. There are two kinds of prompts: **analysis prompts** and **description prompts**. The analysis prompts to be summarised below can be thought of as predicates that the domain engineer applies to phenomena of the domain yielding true, false or undefined answers. The description prompts to be summarised below are applied, by the domain engineer, to phenomena of the domain for which preceding analysis prompts has yielded truth answers. Thus the *domain analysis & description process* alternates between analysis prompts and description prompts. The **domain description method** is here specialised to **manifest domains** [11]. First the domain engineer cum scientist examines a perceived domain phenomena, ϕ: `is_entity`(ϕ), and if **true**, then inquires which of `is_endurant`(ϕ) or `is_perdurant`(ϕ) holds. If `is_endurant`(ϕ) holds then the domain analyser inquires as to whether `is_discrete`(ϕ) or `is_continuous`(ϕ) holds. If `is_discrete`(ϕ) holds then `is_part`(ϕ) holds, otherwise either of `is_material` or `is_component` holds. If `is_part`(ϕ) then either `is_atomic`(ϕ) or `is_composite`(ϕ). If `is_composite`(ϕ) holds then `observe_parts`(ϕ) yields some parts that can now be analysed, eventually leading the domain analyser to conclude that the part ϕ can be described. By applying `observe_part_sorts`(ϕ) to a composite domain δ we then obtain its constituent parts — as exemplified in formula lines 1.–1c. and similarly formula lines 2a.–2b. Some composite parts may be modelled by concrete types: `has_concrete_type`(ϕ) in which case `observe_part_types`(ϕ) will yield those concrete types as exemplified in formula lines 3.–5 , and in formula lines 21 . Once the atomic and composite parts of a domain has been settled their properties: unique identifiers, mereology and attributes can be analysed and described. First their uniqueness: `observe_unique_identifiers`, such as f.ex. illustrated by formula lines 7a.–7b. Once all parts have been identified one can inquire as to their mereology: how parts relate to other parts: if `has_mereology`(ϕ) holds then `observe_mereology`(ϕ) yields which specific other parts, of same or other sorts, such as for example in formula lines 9.–10 or formula lines 31. Finally a last set of properties of parts can be investigated, namely their attributes. Any part, ϕ, may have any number of attributes. The analysis prompt `attribute_names`(ϕ) yields names of attributes. — with the description prompt `observe_attributes`(ϕ) yielding their description — as in formula lines 12a.–12b. or in formula lines 16.–18b.

There are other aspects to the methodology analysing and describing endurants: gaseous or liquid materials being contained in parts, and perdurants actions, events and behaviours. We shall not cover these here, but refer to [10, Manifest Domains: Analysis & Description] and [11, From Domain Descriptions to Requirements Prescriptions].

3.2.8 Domain Science

There are a number of issues that need be researched.

A Prompt Semantics: The analysis and description prompts need be precisely, that is, mathematically defined. Such a semantics is a first step towards securing a foundation for our approach. We refer to [8].

Laws of Domain Descriptions: A semantics of the analysis and description prompts and thus their applications is expected to satisfy the following law: Analysing ($\mathcal{A}$) and/or describing ($\mathcal{D}$) two otherwise unrelated composite parts, p_i and p_j, shall yield the same results whether p_i is treated before p_j or vice-versa: $\mathcal{A}(p_i);\mathcal{A}(p_j)$ and $\mathcal{A}(p_j);\mathcal{A}(p_i)$, respectively $\mathcal{D}(p_i);\mathcal{D}(p_j)$ and $\mathcal{D}(p_j);\mathcal{D}(p_i)$. There are many others such laws.

Laws of Domains: Given an appropriate domain description it should be possible to prove certain laws about that domain. **An example:** Assume a railway system with trains operating according to a timetable that prescribes train departures from and arrivals at any station according to a 24 hour cycle, and assume that all trains function precisely. Now we would expect the following law to hold over any 24 hour period: The of trains *arriving* at a station, minus the number of trains *ending* their journey at that station, plus number of trains *starting* their journey at that station, equals the number of trains *leaving* that station •

• • •

Physics is characterised by its laws. So should man-assisted domains. A proper theory of domain description should invite domain laws to be identified and proved. There is a rich world *"out there"*.

3.2.9 What Can Be Described ?

Even if we limit ourselves to physically manifest domains [11], that is, entities that we can observe, i.e., see, in cases even touch, there are such which we do not yet know how to describe objectively, that is, mathematically. Moreover, we cannot give a precise delineation of which domains, or aspects of domains, are describable.

An example: We have described aspects of a pipeline system, Sect. 3.2.6. We have even postulated (implementable) functions for observing the flow and leaks of the material (oil, gas, or other) conducted by pipeline units. We also know, but do not show, how to formalise the fluid dynamics of these flows, namely in terms of partial differential equations (PDEs) based on Bernoulli and Navier–Stokes models through individual pipeline units. But we have yet to show how to combine our "discrete mathematics" models with hose of fluid dynamics. One problem here is that our discrete mathematics descriptions model an infinite variety of pipelines, that is, arbitrary compositions of pipeline units, whereas, conventionally, PDEs, model the dynamics only of specific, single units. It has been suggested[9] that perhaps the Wiener–Feynman–Dirac–Wheeler concept of *Path Integrals*.[10] may be a way to solve the problem •

Our domain models are just abstractions ! One cannot expect any domain description to "completely" model a domain. There are simply too many properties to describe. And there are domain properties that we can informally describe in words, but cannot yet formalise. Domain description is (therefore) a matter of choice, of abstraction level and of what to include in the description and what to leave out !

3.3 Requirements Engineering

We would not advocate the *TripTych* to software development unless we had a method for "deriving" requirements from domain descriptions. And from formal requirements prescriptions we know how to design software such that $\mathcal{D},\mathcal{S} \models \mathcal{R}$, that is: the $\mathcal{S}$oftware can be proved correct — in the context of the $\mathcal{D}$omain — with respect to the $\mathcal{R}$equirements.

3.3.1 Three Kinds of Requirements

Our approach to the "derivation" of requirements is based on the following decomposition of requirements into three kinds: *domain requirements*, *interface requirements* and *machine requirements* where the *machine* is the hardware and software to be developed

3.3.2 Domain Requirements

By *domain requirements* we shall understand such requirements that can be expressed sôlely using terms of the domain that is, terms defined in the domain description.

[9] Jakob Bohr, Technical University of Denmark

[10] The path integral formulation of quantum mechanics is a description of quantum theory which generalizes the action principle of classical mechanics. It replaces the classical notion of a single, unique trajectory for a system with a sum, or functional integral, over an infinity of possible trajectories to compute a quantum amplitude `wikipedia.org/wiki/Path_integral_formulation`.

The "derivation"[11] of domain requirements prescriptions from domain descriptions is governed by a set of "derivation" operations. Examples of these 'derivation' operations are: *projection*, *instantiation*, *determination*, *extension* and *fitting*.

Projection means that we remove from the evolving requirements prescription those entity descriptions of the domain which are not to be considered when (further) prescribing the requirements. **An example:** From the example of the road net and traffic system we remove the vehicles and the monitor •

Instantiation means that we concretise, i.e., prescribe "less-abstract", those retained domain phenomena whose concretisation it is suitable to prescribe. **An example:** The general road net is instantiated to a "linear" toll-road system of a sequence of toll-road hubs connected, "up" and "down" the toll-road to neighbouring toll-road hubs, and, by means of toll-road plazas, to a remaining road net • What do we mean by: "it is suitable to prescribe"? Well, first of all, we have to realize the following: requirements must only prescribe what can be computed. That means that entities whose realisability in terms of computable data structure or functions must eventually be so prescribed. Secondly, as requirements prescription may, and normally will proceed in stages, one (i.e., the requirements engineer) may decide to instantiate some entities while leaving other entities "untouched", only to return to the concretisation of these n a later stage. And so forth. It is all a matter of style and taste!

Determination means that there may be entities, i.e., endurants or perdurants, that are described to be non-deterministic in the domain but which, after projection and instantiation need be prescribed to be "less non-deterministic". **An example:** Whereas hubs in general allow traffic from any link incident upon that hub to any links emanating from that hub but so that signaling, as expressed in the hub states, may, at times, prevent some emanating links to be accessible from some incident links; a toll-road hub, in order to be an appropriate toll-road, must allow for free flow from any incident link to any emanating link •

Extension typically means that there may be entities that were "hitherto" not computationally feasible, but where new technologies or higher labour costs mandate their feasibility — thus making way for introducing these mew technologies into a this 'extended' domain. **An example** is that of the electronic sensing of vehicles entering or leaving a toll-road — thus enabling *"road pricing"* •

Fitting is necessitated when two or more requirements projects based on "the same" domain, and with "overlapping domain coverage" need be "harmonised". **An example:** One set of requirements are being prescribed for a *road state-of-repair and maintenance facility,* another set of requirements are being prescribed for a *road pricing system*. Now they must both rely on some sort of representation of the same road net •

3.3.3 Interface Requirements

By *interface requirements* we shall understand such requirements that can be expressed only using terms both of the domain and of the machine.

In order to structure the interface requirements we introduce a notion of *shared phenomena* whether endurants or perdurants. If a phenomenon is present in the domain and if it is also to be present in the machine to be designed then that phenomenon is said to be shared. As a result we structure interface requirements prescriptions around *shared endurants*, *shared actions*, *shared events* and *shared behaviours*.

Shared endurants pose two "problems" the initialisation of endurant data structures and their values, and the regular access to and update of endurant data. Both must be prescribed. Usually both require the interaction between the domain and the machine. **An example:** Road nets are shared between the domain and the machine. Initially all hubs and all links need be structured in some data structure, say a database. The shared endurant requirements must now specify which, usually composite database operations are to be used in establishing the database, and which are to be used in accessing and updating the endurants.

Shared actions imply an interaction between between the domain and the machine. That interaction is typically manifested by interaction between either humans of the domain or physical domain entities and the machine **Example: Human/Machine Interaction:** The payment of a road price fee today involves a human (say, with a credit card) and the machine, checking and accepting or rejecting the credit card, etcetera • **Example: Machine/Machine Interaction:** The electronic recording (within the machine) of a vehicle passing a toll-gate barrier (another part of the machine) and the vehicle itself (another machine, external to required machine) •

And so on, for *shared events* and *shared behaviours*.

[11] We put 'derivation' in double quotes because we do not mean 'automatic' derivation.

3.3.4 Machine Requirements

By *machine requirements* we shall understand such requirements that can be expressed sôlely using terms of the machine. Since that is the case: no "mention" of the domain in the machine requirements we shall omit covering this field.

3.4 Discussion

We have suggested that there are a set of principles and techniques for "deriving" a major set of requirements from domain descriptions. This, then, is an argument for taking domain modelling serious: there are principles and techniques for bringing you from domain descriptions to requirements prescriptions and from there on to software design. We refer to [10, 2015] for details.

4 Are We Studying the Right Things ?

We claim to have justified our claim that $\mathcal{S}$*oftware* must be *designed* on the basis of $\mathcal{R}$*equirements prescriptions* that have been "derived" from $\mathcal{D}$*omain descriptions*, all of them formally. In this way we can secure that software fulfill users'/customers' expectations since the requirements are strongly related to the domain and is correct: $\mathcal{D},\mathcal{S} \models \mathcal{R}$.

It is the only way in which we can see these two, *expectations* and *correctness*, fulfilled.

4.1 Papers on Domain Science & Engineering

I mention but a few of my earlier papers related to domain science & engineering. In chronological order. For a comprehensive introduction to Domain Science & Engineering and to a novel approach to Requirements Engineering I refer to [11, 10] respectively.

[5, *Domain Engineering, 2008*] treats and aspect of domain modelling referred to as *domain facets*. We expect to revise [5].

[6, *Domains: Their Simulation, Monitoring and Control, 2011*]. The concepts of simulation, monitoring and simulation are analysed in the light of the domain–requirements–design *TripTych*.

[7, *A Rôle for Mereology in Domain Science and Engineering, 2009*]. Stanisław Leśhniewski's replacement of Bertrand Russells set theory axiomatisation is reviewed amd it is shown how part/sub-part relations can interpreted as a reation between (Hoare) CSP-processes.

[9, *Domain Engineering – A Basis for Safety Critical Software. 2014*]. Issues of *system safety criticality* that can be considered already before requirements engineering are here seen in the light of domain engineering.

[11, *Manifest Domains: Analysis & Description, 2014*] is the definitive paper on domain analysis and description.

[10, *From Domains to Requirements — A Different View of Requirements Engineering, 2015*] is the definitive paper on "derivation" of requirements prescriptions from domain descriptions. It is a complete rewrite of [4, From Domains to Requirements] and represents a complete rethinking of that paper.

4.2 A Research and Experimental Engineering Programme

In the papers on which the current paper is based a number of open problems have been identified.

4.2.1 The Mathematics of Analysis & Description Prompts

In [11, *Domain Analysis: Endurants – An Analysis & Description Process Model*] we present a formal semantics of the analysis and description process. In [10, *From Domain Descriptions to Requirements Prescriptions — a Different Approach to Requirements Engineering*] we present a $\boxed{\dots}$ The study of this area is elusive.

4.2.2 Analysis & Description Calculi for Other Domains

The analysis and description calculus of this paper appears suitable for manifest domains. For other domains other calculi appears necessary. There is the introvert, composite domain of systems software: operating systems, compilers, database management systems, Internet-related software, etcetera. The classical computer science and software engineering disciplines related to these components of systems software appears to have provided the necessary analysis and description "calculi." There is the domain of financial systems software accounting & bookkeeping, banking systems, insurance, financial instruments handling (stocks, etc.), etcetera. Etcetera. For each domain characterisable by a distinct set of analysis & description calculus prompts such calculi must be identified. It seems straightforward: to base a method for analysing & describing a category of domains on the idea of prompts like those developed in this paper.

4.2.3 On Domain Description Languages

We have in this paper expressed the domain descriptions in the `RAISE` [27] specification language `RSL` [26]. With what is thought of as basically inessential, editorial changes, one can reformulate these domain description texts in either of `Alloy` [30] or `The B-Method` [1] or `VDM` [17, 18, 23] or `Z` [36]. One could also express domain descriptions algebraically, for example in `CafeOBJ` [25, 22, 24, 20]. The analysis and the description prompts remain the same. The description prompts now lead to `CafeOBJ` texts.

We did not go into much detail with respect to perdurants, let alone behaviours. For all the very many domain descriptions, covered elsewhere, `RSL` (with its `CSP` sub-language) suffices. But there are cases where we have conjoined our `RSL` domain descriptions with descriptions in `Petri Nets` [34] or `MSC` [29] (Message Sequence Charts) or `StateCharts` [28]. Since this paper only focused on endurants there was no need, it appears, to get involved in temporal issues. When that becomes necessary, in a study or description of perdurants, then we either deploy `DC: The Duration Calculus` [37] or `TLA+: Temporal Logic of Actions` [32].

4.2.4 Commensurate Discrete and Continuous Models

The pipeline example hinted at co-extensive descriptions of discrete and continuous behaviours, the former in, for example, `RSL`, the latter in, typically, the calculus mathematics of partial different equations (`PDE`s). The problem that arises in this situation is the following: there will be, say variable identifiers, e.g., x, y, $\ldots$, z which in the `RSL` formalisation has one set of meanings, but which in the `PDE` “formalisation” has another set of meanings. Current formal specification languages[12] do not cope with continuity. Some research is going on. But to substantially cover, for example, the proper description of laminar and turbulent flows in networks (e.g., pipelines) requires more substantial results.

4.2.5 Interplay between Parts and Materials

The pipeline example revealed but a small fraction of the problems that may arise in connection with modeling the interplay between parts and materials. Subject to proper formal specification language and, for example `PDE` specification we may expect more interesting laws, as for example those of pipeline flows and even proof of these as if they were theorems. Formal specifications have focused on verifying properties of requirements and software designs. With co-extensive (i.e., commensurate) formal specifications of both discrete and continuous behaviours we may expect formal specifications to also serve as bases for predictions.

4.2.6 The Mathematics of Domain-to-Requirements Operators

In [10, *From Domain Specifications to Requirements Prescriptions – A Different View of Requirements Engineering*][13] we postulate that certain properties hold between domain requirements prescriptions “before”and “after” the application of the domain-to-requirements operations: *projection*, *instantiation*, *determination*, *extension* and *fitting*. These postulated properties need be studied further.

4.2.7 Further Work on Domain-to-Requirements and Interface Techniques

In [10, *From Domain Specifications to Requirements Prescriptions – A Different View of Requirements Engineering*] we have shown a number of techniques for domain-to-requirements operations, in particular those that yield domain requirements. In [10] we also show some techniques that pertain to interface requirements, but it seems more study is required.

[12] `Alloy` [30], `Event B` [1], `RSL` [26], `VDM-SL` [17, 18, 23], `Z`, etc.

[13] [10] is a complete rewrite/rethinking of [4].

4.3 Tony Hoare's Summary on 'Domain Modeling'

In a 2006 e-mail, in response, undoubtedly to my steadfast, perhaps conceived as stubborn insistence, on domain engineering, Tony Hoare summed up his reaction to domain engineering as follows, and I quote[14]:

"There are many unique contributions that can be made by domain modeling.

1 The models describe all aspects of the real world that are relevant for any good software design in the area. They describe possible places to define the system boundary for any particular project.

2 They make explicit the preconditions about the real world that have to be made in any embedded software design, especially one that is going to be formally proved.

3 They describe the whole range of possible designs for the software, and the whole range of technologies available for its realisation.

4 They provide a framework for a full analysis of requirements, which is wholly independent of the technology of implementation.

5 They enumerate and analyse the decisions that must be taken earlier or later in any design project, and identify those that are independent and those that conflict. Late discovery of feature interactions can be avoided."

4.4 Are We Studying the Right Things ?

By **computer science** we understand the study and knowledge about the phenomena that can "exist inside" computers. By **computing science** we understand the study and knowledge about how to construct those phenomena.

If we accept the *TripTych* dogma of basing software design on precise requirements prescriptions which are based on precise domain descriptions, then training, teaching and research in computer and computing science must be revised.

5 Acknowledgements

I thank Prof. Jens Knoop for challenging me to present this paper.

6 Bibliography

6.1 References

[1] J.-R. Abrial. The B Book: Assigning Programs to Meanings *and* Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge, England, 1996 and 2009.

[2] D. Bjørner. Programming Languages: Formal Development of Interpreters and Compilers. In *International Computing Symposium 77 (eds. E. Morlet and D. Ribbens)*, pages 1–21. European ACM, North-Holland Publ.Co., Amsterdam, 1977.

[3] D. Bjørner. Domain Models of "The Market" — in Preparation for E–Transaction Systems. In *Practical Foundations of Business and System Specifications (Eds.: Haim Kilov and Ken Baclawski)*, The Netherlands, December 2002. Kluwer Academic Press. Final draft version.

[4] D. Bjørner. From Domains to Requirements. In *Montanari Festschrift*, volume 5065 of *Lecture Notes in Computer Science (eds. Pierpaolo Degano, Rocco De Nicola and José Meseguer)*, pages 1–30, Heidelberg, May 2008. Springer.

[5] D. Bjørner. Domain Engineering. In P. Boca and J. Bowen, editors, *Formal Methods: State of the Art and New Directions*, Eds. Paul Boca and Jonathan Bowen, pages 1–42, London, UK, 2010. Springer.

[6] D. Bjørner. Domains: Their Simulation, Monitoring and Control – A Divertimento of Ideas and Suggestions. In *Rainbow of Computer Science, Festschrift for Hermann Maurer on the Occasion of His 70th Anniversary.*, Festschrift (eds. C. Calude, G. Rozenberg and A. Saloma), pages 167–183. Springer, Heidelberg, Germany, January 2011.

[14] E-Mail to Dines Bjørner, July 19, 2006

[7] D. Bjørner. *A Rôle for Mereology in Domain Science and Engineering*. Synthese Library (eds. Claudio Calosi and Pierluigi Graziani). Springer, Amsterdam, The Netherlands, October 2014.

[8] D. Bjørner. Domain Analysis: Endurants – An Analysis & Description Process Model. In S. Iida, J. Meseguer, and K. Ogata, editors, *Specification, Algebra, and Software: A Festschrift Symposium in Honor of Kokichi Futatsugi*. Springer, May 2014.

[9] D. Bjørner. Domain Engineering – A Basis for Safety Critical Software. Invited Keynote, ASSC2014: Australian System Safety Conference, Melbourne, 26–28 May, December 2014.

[10] D. Bjørner. From Domain Descriptions to Requirements Prescriptions – A Different Approach to Requirements Engineering. *Submitted for consideration by Formal Aspects of Computing*, 35 pages. 2016.

[11] D. Bjørner. Manifest Domains: Analysis & Description. *Expected published by Formal Aspects of Computing*, 44 pages. 2016.

[12] D. Bjørner. *[14] Chap. 7: Documents – A Rough Sketch Domain Analysis*, pages 179–200. JAIST Press, March 2009.

[13] D. Bjørner. *[14] Chap. 9: Towards a Model of IT Security — – The ISO Information Security Code of Practice – An Incomplete Rough Sketch Analysis*, pages 223–282. JAIST Press, March 2009.

[14] D. Bjørner. *Domain Engineering: Technology Management, Research and Engineering*. A JAIST Press Research Monograph # 4, 536 pages, March 2009.

[15] D. Bjørner and K. Havelund. 40 Years of Formal Methods — 10 Obstacles and 3 Possibilities. In *FM 2014, Singapore, May 14-16, 2014*. Springer, 2014. Distinguished Lecture.

[16] D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer, 1978. This was the first monograph on $Meta\text{-}IV$.

[17] D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer, 1978.

[18] D. Bjørner and C. B. Jones, editors. *Formal Specification and Software Development*. Prentice-Hall, 1982.

[19] D. Bjørner and O. N. Oest, editors. *Towards a Formal Description of Ada*, volume 98 of *LNCS*. Springer, 1980.

[20] CafeOBJ. http://cafeobj.org/, 2014.

[21] G. Clemmensen and O. Oest. Formal specification and development of an Ada compiler – a VDM case study. In *Proc. 7th International Conf. on Software Engineering, 26.-29. March 1984, Orlando, Florida*, pages 430–440. IEEE, 1984.

[22] R. Diaconescu and K. Futatsugi. *CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*. AMAST Series in Computing, 6. World Scientific, Singapore, 1998.

[23] J. Fitzgerald and P. G. Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 1998. ISBN 0-521-62348-0.

[24] K. Futatsugi, D. Găină, and K. Ogata. Principles of proof scores in CafeOBJ. *Theor. Comput. Sci.*, 464:90–112, 2012.

[25] K. Futatsugi and A. Nakagawa. An overview of CAFE specification environment - an algebraic approach for creating, verifying, and maintaining formal specifications over networks. In *Proc. of 1st International Conference on Formal Engineering Methods (ICFEM '97), November 12-14, 1997, Hiroshima, JAPAN*, pages 170–182. IEEE, 1997.

[26] C. W. George, P. Haff, K. Havelund, A. E. Haxthausen, R. Milne, C. B. Nielsen, S. Prehn, and K. R. Wagner. *The RAISE Specification Language*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1992.

[27] C. W. George, A. E. Haxthausen, S. Hughes, R. Milne, S. Prehn, and J. S. Pedersen. *The RAISE Development Method*. The BCS Practitioner Series. Prentice-Hall, Hemel Hampstead, England, 1995.

[28] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[29] ITU-T. CCITT Recommendation Z.120: Message Sequence Chart (MSC), 1992, 1996, 1999.

[30] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge, Mass., USA, April 2006. ISBN 0-262-10114-9.

[31] C. B. Jones. *Software Development: A Rigorous Approach*. Prentice-Hall, 1980.

[32] L. Lamport. *Specifying Systems*. Addison–Wesley, Boston, Mass., USA, 2002.

[33] O. Oest. VDM From Research to Practice. In H.-J. Kugler, editor, *Information Processing '86*, pages 527–533. IFIP World Congress Proceedings, North-Holland Publ.Co., Amsterdam, 1986.

[34] W. Reisig. *Petrinetze: Modellierungstechnik, Analysemethoden, Fallstudien*. Leitfäden der Informatik. Vieweg+Teubner, 1st edition, 15 June 2010. 248 pages; ISBN 978-3-8348-1290-2.

[35] J. Woodcock, J. B. P.G. Larsen, and J. Fitzgerald. Formal Methods: Practice and Experience. *ACM Computing Surveys*, 41(4):19, 2009.

[36] J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.

[37] C. C. Zhou and M. R. Hansen. *Duration Calculus: A Formal Approach to Real–time Systems*. Monographs in Theoretical Computer Science. An EATCS Series. Springer–Verlag, 2004.

Alea Reactive Dataflow: GPU Parallelization Made Simple

presented at the 2014 SPLASH conference REBLS workshop without formal proceedings

Luc Bläser
University of Applied Sciences
Rapperswil
Institute for Software
lblaeser@hsr.ch

Daniel Egloff
QuantAlea Inc. Zurich
daniel.egloff@quantalea.net

Philipp Kramer
University of Applied Sciences
Rapperswil
Institute for Software
pkramer@hsr.ch

Xiang Zhang
QuantAlea Inc. Zurich
xiang.zhang@quantalea.ch

Abstract

Making effective use of the GPU parallel power requires relatively complex and tedious work: Understandably, most programmers spare the efforts. The Alea reactive dataflow programming model now aims to substantially lower this threshold by simplifying GPU parallelization quite radically. Programs are described as data that is asynchronously propagated through a graph of operations, each typically predestined for vector parallelization. Programmers do no longer need to write GPU-specific code but instead leave the GPU-parallelization to the runtime system. Due to the declarative and reactive paradigm, operations can be easily scheduled as parallel streams on a GPU with minimum memory copying overheads.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Concurrent programming structures

General Terms Languages

Keywords GPU; parallelization; reactive; dataflow

1. Introduction

For many programmers, the threshold for engaging GPU parallelization is too high. In order to make adequate use of the many cores of a GPU, several obstacles need to be taken: (1) Algorithms need to be tailored for vector parallelization since the cores are de facto per-element views of vector-parallel instructions. (2) The parallel implementation is based on a rather low-level machine-centric programming models such as CUDA [1], OpenCL [2], or other alternatives [3, 4]. (3) Integrating the typical C technology stack into a managed environment, such as .NET, necessitates extra workarounds. Therefore, GPU parallelization is unfortunately often perceived as too difficult, too costly and offering only a marginal benefit.

Several cross-platform frameworks support GPU parallel programming in managed runtimes, such as in Java [6, 7] or .NET [5, 8, 9]. The programming abstractions, however, essentially remain at the same low level of CUDA or OpenCL. More elegant integrations have been proposed but they usually lack generality, e.g. a LINQ-integration [10, 11] only supports a limited set of query operations. Dataflow programming models for GPU are more general: Xcelerit [12], PTask [13], and FastFlow [14] follow this approach but have the drawback that programmers typically have to implement custom nodes since the model has no or only fixed predefined operations. This is where more low-level and tedious programming is again involved. We also believe that substantial benefits can be gained if dataflow would become more reactive, i.e. fully asynchronous and ready to process sequences of inputs. For a more detailed analysis, see the discussion of related work in Section 4.

Our goal is to radically simplify GPU parallelization while still retaining expressiveness and efficiency. For this reason, we have developed *Alea reactive dataflow*, a programming model based on .NET. A computation is described as data propagated through a directed graph of operations. The propagation is asynchronous, reactive and push-based, while operations are typically vector-parallelizable and generic. Programmers can easily define computations without writing GPU code.

The runtime system takes care of the efficient parallelization on GPUs, by streaming operations, configuring launches and minimizing copying between CPU and GPU memory. The runtime system as well as the implementation of operations guarantee memory safety. Although we currently focus on GPUs, the model could be equally applied to general heterogeneous distributed parallelization.

The remainder of this paper is structured as follows. Section 2 introduces the programming model. Section 3 briefly outlines the current runtime system. Section 4 discusses related works. Section 5 finally draws a conclusion.

2. Programming Model

Alea reactive dataflow programs are defined by connecting operations to form a directed graph. Computations are triggered by feeding input to operations. This implies a chain of reactions: Operations execute asynchronously whenever sufficient input is present and thereby produce output passed to subsequent operations. To obtain results, output can be observed from any operation. In the

CONF 'yy, Month d–d, 20yy, City, ST, Country.
Copyright © 20yy ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

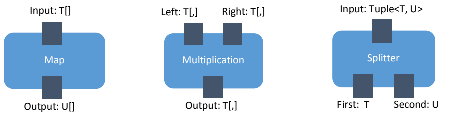
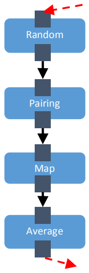
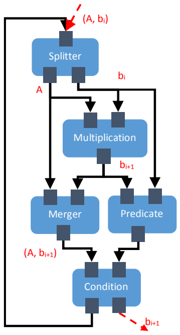


Figure 1. Three operations with input and output ports.



```
var random = new Random<float>();
var pairing = new Pairing<float>();
var map = new Map(p =>
   p.X * p.X + p.Y * p.Y <= 1
   ? 1 : 0);
var average = new Average<int>();

random.Output.ConnectTo(pairing.Input);
pairing.Output.ConnectTo(map.Input);
map.Output.ConnectTo(average.Input);

average.Output.OnReceive(x =>
           Console.WriteLine(x * 4));

random.Input.Send(1000);
random.Input.Send(1000000);
```

Figure 2. Monte Carlo Pi approximation dataflow graph.



```
var source = new Splitter
                   <Matrix, Vector>();
var mult = new Multiplication
                   <Matrix, Vector>();
var cmp = new Predicate<Vector>
        ((a, b) => Abs(a - b) > 1E-6);
var next = new Merger<Matrix, Vector>();
var cond = new Condition<Vector>();

source.First.ConnectTo(mult.First);
source.Second.ConnectTo(mult.Second);
source.First.ConnectTo(next.First);
mult.Output.ConnectTo(next.Second);
mult.Output.ConnectTo(cmp.First);
source.Second.ConnectTo(cmp.Second);
next.Output.ConnectTo(cond.PassThrough);
cmp.Output.ConnectTo(cond.Criterion);
cond.True.ConnectTo(source.Input);

cond.False.OnReceive(Console.WriteLine);
source.Send(new Tuple(A, b0));
```

Figure 3. Markov chain steady state as dataflow graph.

following subsections, we explain the elementary concepts of the programming model, accompanied by two exemplary application cases.

2.1 Operations

An *operation* represents a self-contained unit of calculation that has a set of input ports and a set of output ports. A *port* denotes a stream of data of a defined type. The stream can be infinite with data arriving in arbitrary intervals. When data is present at a defined set or subset of the input ports, the operation consumes this data as input, performs a calculation to produce data as output for a set or subset output ports. Input is processed in the order as it arrives, triggering output in the corresponding order, i.e. later input cannot result in earlier output. However, data can arrive at each port at different time intervals; they are not mutually synchronized.

Figure 1 depicts operations, with input ports at the top border and output ports at the bottom border. Each port is specified with a name and the type of the data. An operation is an instance of a particular class, implementing the operation. The operation determines which input ports are required to trigger a calculation, e.g. `Multiplication` requires data at both input ports to trigger. Analogously, the operation also defines to which output ports data is passed, e.g. `Splitter` yields data at both output ports for each input.

Operations can feature multiple implementations for different processor architectures, such as GPUs or CPUs, see Section 3. To be suited for GPUs, operations typically implement a massively vector-parallel (SIMD) calculation per input, e.g. `Map` transforms an array of elements. Many operations are generic, i.e. only provide partial implementation skeleton to be completed by a delegate/lambda at construction time, e.g. the element-wise map function delegate of the `Map` operation. This enables relatively high expressiveness despite a fixed set of prefabricated operation classes. Internally, operations can be stateless or stateful, i.e. work with or without a state that is stored between executions.

2.2 Graphs

Operations can be interconnected to form a *graph*. The output port of a preceding operation can be connected to one or multiple input ports of a succeeding operation, provided that the ports have the same type. Whenever data is passed to an output port, the data becomes available at all connected input ports. Multiple output ports may be also connect to the same input port, if the types match, using an arbitrary order to merge the data of multiple output ports into a common input port.

Figure 2 outlines a graph for a Monte Carlo Pi approximation. Figure 3 shows a graph for the iterative computation of the steady state in a Markov chain, based on the iterative formula $b_{i+1} = Ab_i$ until $b_{i+1} = b_i$. `Splitter` and `Merger` are used to synchronize A and b input, in the case of concurrent processing of multiple Markov chain inputs.

Passing is asynchronous, i.e. an operation can produce data to an output port, without awaiting the consumption of the data by any other connected operations. Operations adhere to the principle that passed data is immutable. Data can thus be passed by copying or by referencing. If an arbitrary merge order is inappropriate, dedicated operations may be used to join multiple data streams. Graphs can have cyclic connections, such as for iterative or continuous computations (e.g. Figure 3). Ports can also have no connections: they may be unused or serve for external sending or reception, as explained in the next subsection.

2.3 Dataflow

A *dataflow* is the propagation of data through the graph. Data can be sent to any input port. Sending is asynchronous, i.e. does not block. Multiple data can also be sent at the same time to the same input port, in which case no order is postulated for the data. Conversely, data can also be received from any output port by registering delegates that are asynchronously invoked whenever output data is produced at that port. Figure 2 and 3 also demonstrate how data is sent to input ports and received from output ports (highlighted in red font).

The reception delegates are executed by arbitrary threads, i.e. multiple output data can be processed concurrently. If multiple delegates are registered for an output port, all are invoked in arbitrary order or possibly concurrently. Data streams require no explicit termination but represent a conceptually infinite sequence of data.

2.4 Short Notation

A shortcut fluent-style notation can be used for the graph and dataflow definition, see Figure 4. The selection of input and output

```
var random = new Random<float>();
random
  .Pairing()
  .Map(p => p.X * p.X + p.Y * p.Y <= 1 ? 1 : 0)
  .Average()
  .OnReceive(a => Console.WriteLine(a * 4));
random.Send(1000);
random.Send(1000000);
```

Figure 4. Short notation for the Monte Carlo Pi example.

port is thereby implicit if the operation has a single input or output port, respectively.

3. Runtime System

The dataflow runtime support is realized by two components: a scheduler and the internal implementations of operations.

Operations implement a function for determining when sufficient input is available to trigger the calculation. Moreover, an operation provides one or multiple mappings to defined processor architectures, such as CPU and GPU. The GPU mapping resembles the standard CUDA model [1], however type-safely integrated into .NET. As for generic operations, the concrete delegate .NET IL code is gathered and translated to CUDA code at runtime or at compile-time, and eventually fused into the operation's CUDA kernel. The GPU mapping of an operation additionally defines a script of specific malloc/launch commands as an execution plan to happen in the future. At the planning time, the script has only restricted information about the data to be processed, i.e. only knows scalar values and the sizes of input blocks to make decisions for optimal kernel launch configurations. The CPU mapping can be directly executed by the .NET TPL [15]. In contrast to GPU mappings, a CPU implementation can be stateful, i.e. carry state over calculation by defining their own CPU-side synchronization on that state.

Programmers can implement custom operations, by providing the mapping for CPU and/or CUDA. This certainly requires more expert knowledge in GPU parallelization. However, we aim to provide a good base functionality supplying a well-selected set of generic operations, such that users usually do not need to implement custom operations.

The scheduling is currently realized for hybrid CPU and single GPU execution. For an input, the scheduler collects the largest non-cyclic sub-graph of GPU-implemented operations to start these operation in one stream. Memory copying is only necessary and performed for transitions between CPU and GPU operations or when the host program sends data to or receives data from GPU operations. As transmitted data must be immutable, it can be shared or copied. Deallocation of GPU memory is automatically managed by the scheduler and not within the operation implementations. The scheduler disposes GPU memory blocks when no longer used by a running operation or contained in a data stream.

The dataflow system uses Alea cuBase [8] as the underlying engine for the CUDA runtime and compilation support within .NET.

4. Related Work

Our model is strongly inspired by Rx.NET [16, 17] and TPL dataflow [18]. These models are however not designed for GPUs, as the blocks are generally unsuited for vector parallelization. A further significant difference is that we support multiple input and output ports. This permits the design of arbitrary well-controlled mergers or splitters. In the TPL dataflow for example, splitting and merging can only be controlled to a limited degree, by filtering messages or using batch/join blocks with specific merge pattern. We also abandon the concept of explicit termination of a stream.

Several frameworks improve cross-platform GPU parallelization, e.g. for Java [6, 7] or .NET [5, 8, 9]. However, the majority essentially exposes the same low-level programming model. Programmers are still bothered by technical artefacts, such as writing SIMD-kernels, copying between CPU and GPU memory, wrapping code in special classes, dealing with launch configurations, thread block ids etc. Notable simplification are achieved by more abstract models, such as translating .NET LINQ expressions to GPU parallel code [10, 11]. However, the expressiveness of this approach is inherently limited by the fixed set of LINQ query functions, basically being projection, mapping, filtering, ordering, and grouping.

Dataflow models allow the composition of parallel operations by minimizing memory transfers. Xcelerit [12], PTask [13], and FastFlow [14] are all based on this paradigm, to enable heterogeneous parallel computing in particular also for GPUs. These system still do not go as far as desired: A created graph essentially serves a single computation and/or synchronous invocation from the host side limits concurrency. In combination with a reactive concept, their practicability could be raised, i.e. by allowing the same graph to asynchronously process a conceptually infinite sequence of inputs, sent in arbitrary intervals. In contrast to the aforementioned systems, we also support generic operations, i.e. operations that implement a partial algorithm skeleton and are completed by a user-specific delegate/lambda/functor upon creation. This naturally requires cross-compilation of host code to the GPU platform.

5. Conclusions

The Alea reactive dataflow programming model enables simple but powerful GPU parallelization in .NET. Due to the descriptive paradigm, programmers are liberated from writing explicit low-level GPU code. This promotes fast and condensed program formulation, while the scheduler enables efficient and memory-safe execution behind the scenes. The reactive push-based paradigm makes the model particularly general, i.e. supports cycles, infinite stream of input delivered in arbitrary intervals. Naturally, the usefulness of the model stands and falls with the set of operations that is available. Generic operations provide a substantial step in this direction, such that programmers usually do not need to implement custom operations. Our work is still in progress: In the future, we plan to enhance the scheduler for the support of multiple GPUs and cluster distribution, as well as for further optimizations. Moreover, we aim to continuously extend the generic operation catalogue.

Addendum

This research is funded by the Swiss National Commission of Technology and Innovation (CTI), project number 16130.2. All trademarks, trade names etc. are the property of their respective owners.

References

[1] Nvidia Inc. *CUDA C Programming Guide*. Version 6.0, `http://docs.nvidia.com/cuda/cuda-c-programming-guide`, accessed 2014-08-25.

[2] Khronos Group. *The Open Standard for Parallel Programming of Heterogeneous Systems*. OpenCL 2.0, `https://www.khronos.org/opencl`, accessed 2014-08-25.

[3] Microsoft Inc. *C++ Accelerated Massive Parallelism (C++ AMP)*. `http://msdn.microsoft.com/en-us/library/hh265136.aspx`, accessed 2014-08-25.

[4] OpenACC. *The OpenACC Application Programming Interface*. Version 1.0, 2011, `http://www.openacc.org`, accessed 2014-08-25.

[5] *Cudafy.NET*. `http://cudafy.codeplex.com`, accessed 2014-08-25.

[6] P. C. Pratt-Szeliga, J. W. Fawcett, and Roy D. Welch. *Rootbeer: Seamlessly using GPUs from Java*. IEEE 9th International Conference

on High Performance Computing and Communication 2012 & IEEE 14th International Conference on on Embedded Software and Systems (HPCC-ICESS), 2012. IEEE, pp. 375-380, 2012.

[7] Y. Yonghong, M. Grossman, and V. Sarkar. *JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA*. Euro-Par 2009 Parallel Processing. Springer, 887-899, 2009.

[8] QuanAlea Inc. *Alea cuBase*. `https://www.quantalea.net`, accessed 2014-08-25.

[9] G. Cocco. *FSCL Compiler*. `http://fscl.github.io/FSCL.Compiler`, accessed 2014-08-25.

[10] Nessos, *GPU LINQ*. `https://github.com/nessos/GpuLinq`, accessed 2014-08-25.

[11] C. J. Rossbach, Y. Yu, J. Currey, J. P. Martin, and D. Fetterly. *Dandelion: A Compiler and Runtime for Heterogeneous Systems*. In Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13). Nov. 2013.

[12] J. Lotze, P. D. Sutton, and H. Lahlou. *Many-Core Accelerated LIBOR Swaption Portfolio Pricing*. In Companion IEEE High Performance Computing, Networking, Storage and Analysis (SCC), 2012.

[13] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. *PTask: Operating System Abstractions to Manage GPUs as Compute Devices*. In Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11), Oct. 2011.

[14] M. Aldinucci, S. Campa, M. Danelutto, P. Kilpatrick, and M. Torquati. *Targeting Distributed Systems in FastFlow*. In Euro-Par 2012: Parallel Processing Workshops (pp. 47-56). Springer, Jan. 2013.

[15] D. Leijen, W. Schulte, and S. Burckhardt. *The Design of a Task Parallel Library*. In Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'09), Oct. 2009.

[16] Microsoft Inc. *The Reactive Extensions (Rx.NET)*, `http://msdn.microsoft.com/en-us/data/gg577609.aspx`, accessed 2014-08-25.

[17] E. Meijer. *Your Mouse is a Database*. ACM Queue 10(3):20-34, March 2012.

[18] S. Toub. *Introduction to TPL Dataflow*. Microsoft Inc, Apr. 2011.

An Approach for Generalized Reversible Functional Programming

Stefan Bohne[1]
Baltasar Trancón Widemann[2]

[1] senTec Elektronik GmbH, Ilmenau
[2] Fakultät für Informatik und Automatisierung, TU Ilmenau

In software engineering, one often comes across pairs of functions which look “something-like-inverse” to each other; reading and writing a file, sending and receiving data, parsing and pretty-printing. Modern programming languages provide almost no support for keeping these function pairs consistent. The concepts bijection and injection more often than not do not apply and there is no other concept readily available for what it means for these function pairs to be consistent. We make an attempt to categorize different classes of such function pairs and investigate their properties whilst providing the basis for a functional programming language with some yet unseen advantages.

1 Introduction

It is not rare to come across pairs of functions that look like inverses of each other. Parsing a grammar and pretty-printing the corresponding tree is one example. These two functions are usually written independently of each other. A close inspection often reveals that their source code shares a lot of structure. For every case distinction in the parser there tends to be a corresponding case in the pretty-printer. The first mathematical concept that comes to mind here are bijective functions, or injective functions if we allow partiality. Unfortunately, the parser/pretty-printer pair typically is not a partial injection. Whitespace and superfluous parentheses are often dropped from the parsed tree. Yet one can still formulate a consistency requirement: pretty-printing a tree and then parsing the result should always yield exactly the same tree. Reading a file while allowing many old versions of the file format and writing the file in the latest version is another function pair that falls into this category. More examples are constructor/pattern pairs of algebraic data types, operations together with their undo-operation in user interfaces, and conversion between different file formats. Such a function pair is the type of mathematical object which we begin to study in this paper.

Keeping both functions consistent with each other by hand can be very error-prone in a large software project. Thus, it is not surprising that research in the area of reversible computation and bidirectional transformations has already investigated several solutions. In [6], simple bidirectional isomorphisms are composed to construct more complex parser/pretty-printer pairs. Logic programming and Definite Clause Grammars are used in [4] to define reversible grammars. The problem of whitespace is not addressed in these frameworks. The programming languages INV [5] for writing injective programs and Boomerang [1] for writing lenses both use a point-free style to compose complex function pairs, which is unfamiliar to many programmers and not always the best paradigm for a given problem. The programming language Janus [9] is an applicative, imperative, reversible programming language. RFun [10] is an applicative, functional, reversible programming language. Janus and RFun only deal with injective functions.

In this paper, we sketch a programming language for reversible programs, that allows a point-free and an applicative style. To make best use of the point-free style, the programming language also includes syntactical constructions common in functional languages. Moreover, we require that the semantics of the reversible sub-language are such that, when a reversible program is interpreted as an irreversible one, it has exactly the same behavior. In other words, reversible programs should be a subset of irreversible ones. Additionally, we will try to find suitable concepts of programs beyond injective functions that are appropriate for the examples given above.

Note that, we want to specify two functions, as opposed to finding an arbitrary reverse function. A comparison between other techniques and the combinator approach that we employ, can be found in [2]. As this is still much work in progress, our results must be taken tentatively.

1.1 Notation and Assumptions

We write function composition using ";" in 'computer science' order, i.e., $f;g = \lambda x \bullet g\,(f\,x)$. Function application binds strongest, followed by function composition. The identity function on A is denoted by $\mathrm{id}_A : A \to A$ with $\mathrm{id}_A\ x = x$ for all $x \in A$. We will drop type subscripts most of the time. When talking about operators, we use dots to denote the operator itself. For example, $\cdot;\cdot : (A \to B) \times (B \to C) \to A \to C$ is the function composition operator.

We use a partial lambda term of the form $\lambda x \mid P(x) \bullet E(x)$ that defines a function only on those values x where the predicate $P(x)$ is true. The function is undefined on other values, including values on which P is undefined. This notation is inspired by the Z-Notation [8].

We assume all functions to be continuous in the domain theoretic sense. Thus, we also assume a complete partial order ($\sqsubseteq$) with bottom ($\perp$) to exist on all types. Top and Bot denote the type of all values and the type of no values respectively.

We will use bold font to denote reversible functions and operators on reversible functions.

2 Structure of a Reversible Functional Program

As the title of this paper says, we are going to describe an approach for a programming language that is both *functional* and *reversible*. By functional, we mean that a program is a function. By functional programming we mean that a program is constructed by composing functions. By reversible we mean programs can be run in two directions: the normal direction, that assigns an output value to an input value, and the reverse direction, that assigns an input value to an output value.

Remark 1. What we mean by reversible here is different from what is typically referred to as reversible. A reversible function can be applied in reverse on its own output to produce *some* input value, not necessarily the original value. The reverse of functions that do have such a behavior are typically called *inverse*. A better terminology for such functions would be *invertible*.

We always treat both directions in parallel resulting in the following definition. This is also called the combinator approach [2].

Definition 1. *A pair of functions $t = (\overrightarrow{t}, \overleftarrow{t})$ with opposing types, i.e., $\overrightarrow{t} : A \to B$ and $\overleftarrow{t} : B \to A$, is called a* janus. *$\overrightarrow{t}$ is called the* normal direction *and $\overleftarrow{t}$ the* reverse direction *of t. We will sometimes write $t = \begin{pmatrix} \overrightarrow{t} \\ \overleftarrow{t} \end{pmatrix}$ where it improves readability. We write $A \rightleftarrows B$ as an abbreviation for $(A \to B) \times (B \to A)$. The* composition, *$t_1 \mathbf{;} t_2 : A \rightleftarrows C$, of two januses $t_1 : A \rightleftarrows B$ and $t_2 : B \rightleftarrows C$ is defined as $t_1 \mathbf{;} t_2 = (\overrightarrow{t_1} ; \overrightarrow{t_2}, \overleftarrow{t_2} ; \overleftarrow{t_1})$. The* reverse, *$t^\dagger : B \rightleftarrows A$, of a janus $t : A \rightleftarrows B$ is defined as $t^\dagger = (\overleftarrow{t}, \overrightarrow{t})$.*

Our approach can now be phrased: whereas functional programming consists of composing functions, generalized reversible functional programming consists of composing januses.

Remark 2. It is easy to see that janus composition is associative, $(t^\dagger)^\dagger = t$ and $(t_1 \mathbf{;} t_2)^\dagger = t_2^\dagger \mathbf{;} t_1^\dagger$. Let $\mathbf{id}_A = (\mathrm{id}_A, \mathrm{id}_A)$, then januses form a dagger category – thus, the choice of symbol for janus reverse. Together with

$$\begin{aligned} A \otimes B &= A \times B \\ t_1 \otimes t_2 &= (\lambda(a,b) \bullet (\overrightarrow{t_1}\, a, \overrightarrow{t_2}\, b), \lambda(c,d) \bullet (\overleftarrow{t_1}\, c, \overleftarrow{t_2}\, d)) \\ \mathbf{swap}_{A,B} &= (\lambda(a,b) \bullet (b,a), \lambda(b,a) \bullet (a,b)) \\ \mathbf{assoc}_{A,B,C} &= (\lambda((a,b),c) \bullet (a,(b,c)), \lambda(a,(b,c)) \bullet ((a,b),c)) \\ I &= \{()\} \\ \mathbf{right}_A &= (\lambda a \bullet (a,()), \lambda(a,()) \bullet a) \end{aligned}$$

januses become a dagger symmetric monoidal category.

Even though this paper is about generalizing reversible programming beyond injective functions, they are a useful object of investigation for finding what makes reversible programs different from irreversible ones. One important observation is that injective functions cannot throw away information. The prime example for functions that throw away information are the projection functions, specifically $\pi_1 : A \times B \to A$ with $\pi_1\,(a,b) = a$. How could a janus look like whose normal direction is that of a projection function? There does not seem to be a generic way how the B-component can be recomputed from the A-component. Thus, we provide one ourselves.

Definition 2. *The janus constructor* $\mathbf{forget}_{A,B} : (A \to B) \to (A \times B \rightleftarrows A)$ *is defined as* $\mathbf{forget}_{A,B}\, f = (\pi_1, \lambda a \bullet (a, f\, a))$.

Related to not using a variable at all is the issue of using the value of a variable more than once. In the irreversible world, this is represented by the function $\delta_A : A \to A \times A$ with $\delta_A\, a = (a, a)$. In the reverse direction, it is possible that the two copies of the variable have different values. Which do we choose? Should we combine them? Disallow such combinations? And again, it is up to the programmer to decide. $(\mathbf{forget}_{A,A}\ \mathrm{id}_A)^\dagger$ is one way – keep the first value and ignore the second. Keeping the second value can be achieved by using $\mathbf{swap};(\mathbf{forget}\ \mathrm{id})^\dagger$. There also exists a canonical solution if the data type admits equality testing.

Definition 3. *Let* A *be a type equipped with a binary function* $\cdot == \cdot : A \times A \to \mathrm{Bool}$ *with* $(a_1 == a_2) \sqsubseteq (a_1 = a_2)$*. Then the janus* $\mathbf{dup}_A : A \rightleftarrows A \times A$ *is defined as* $\mathbf{dup}_A = (\delta_A, \lambda(a_1, a_2) \mid a_1 == a_2 \bullet a_1)$.

2.1 Janus Classes

What we have described so far, are just arbitrary pairs of functions with reverse type signatures. And actually, **forget** is sufficient to construct any janus from its two directions.

Corollary 1. *Any* $t : A \rightleftarrows B$ *can be decomposed as*

$$t = (\mathbf{forget}_{A,B}\ \overrightarrow{t}\,)^\dagger;\mathbf{swap}_{A,B};\mathbf{forget}_{B,A}\ \overleftarrow{t}\quad .$$

If we assume that the relationship between normal and reverse direction in simple januses is always useful, then the mere way in which complex januses are constructed from simpler januses will likely result in a useful janus. Nonetheless, it is possible – and interesting – to ensure certain consistency conditions.

Figure 1 shows the janus subsets, which we call *janus classes*, that are going to be used in the remainder of this paper.

Class's name	Condition	Abbreviation
inverse	$\overrightarrow{t};\overleftarrow{t} \sqsubseteq \mathrm{id} \wedge \overleftarrow{t};\overrightarrow{t} \sqsubseteq \mathrm{id}$	in
semi-inverse	$\overrightarrow{t};\overleftarrow{t} \sqsubseteq \mathrm{id}$	si
reverse semi-inverse	$\overleftarrow{t};\overrightarrow{t} \sqsubseteq \mathrm{id}$	rs
pseudoinverse	$\overrightarrow{t};\overleftarrow{t};\overrightarrow{t} \sqsubseteq \overrightarrow{t} \wedge \overleftarrow{t};\overrightarrow{t};\overleftarrow{t} \sqsubseteq \overleftarrow{t}$	pi
generic	–	gj
irreversible	$(\overleftarrow{t} = \bot)$	ir

Fig. 1: Janus classes

Inverse januses are similar to partial injective functions. The condition states that, if both directions are defined at a point, information is never lost. The difference to injective functions is that one direction may be defined and returns a value at which the other direction is undefined.

Semi-inverse and reverse semi-inverse januses can be seen as transformations that may lose information only in one direction. It is easy to see that **forget** f is reverse semi-inverse for any f, since the information that is computed by f the reverse direction is simply thrown away in the normal direction. All of the example function pairs given in the introduction are actually either semi-inverse or reverse semi-inverse.

Pseudoinverse januses are named so, because they have a lot in common with Moore–Penrose pseudoinverses of matrices. The condition on pseudoinverse januses can be rewritten into $t;t^\dagger;t \sqsubseteq t$, from which $t^\dagger;t;t^\dagger \sqsubseteq t^\dagger$ follows. Also, the januses $t;t^\dagger$ and $t^\dagger;t$ are (partial) idempotent and self-reverse, which loosely corresponds to being hermitian. Pseudoinverse januses can be seen as those losing information in both directions, but only during the first pass. After a pseudoinverse janus has been applied to a value in one direction, applying it again backward and again forward will yield the same value or will be undefined.

Irreversible 'januses' represent normal functions. They are, on one hand, isomorphic to the januses class given by the condition $\overleftarrow{t} = \bot$, but, on the other hand, they can be seen as the type $A \to B \times \mathrm{Top} \to \mathrm{Bot}$. Thus, they aren't really januses from the typing perspective, but this view will be useful when we define the semantics. They basically are januses of which we promise never to invoke the reverse direction.

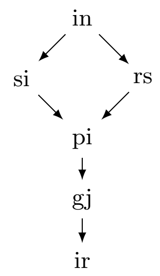
In Figs. 2a and 2b we overload $\cdot;\cdot$ and $\cdot^\dagger$ on janus classes, such that, if t_i is in janus class J_i for $i \in \{1, 2\}$, then $t_1;t_2$ is in janus class $J_1;J_2$ and $t_1^\dagger$ is in janus class $J_1^\dagger$. It is worth noting that inverse, semi-inverse and reverse semi-inverse januses are closed under composition, but pseudoinverse januses are not. Nonetheless, we can still compose pseudoinverse januses with semi-inverse januses from the right and reverse semi-inverse januses from the left.

;	in	si	rs	pi	gj	ir
in	in	si	rs	pi	gj	ir
si	si	si	gj	gj	gj	ir
rs	rs	pi	rs	pi	gj	ir
pi	pi	pi	gj	gj	gj	ir
gj	gj	gj	gj	gj	gj	ir
ir	ir	ir	ir	ir	ir	ir

(a) Composition

J	$J^\dagger$
in	in
si	rs
rs	si
pi	pi
gj	gj
ir	–

(b) Reverse



(c) Inclusion lattice

Fig. 2: Relations between janus classes

The correctness proof for $\cdot^{\dagger}$ is straight-forward. The correctness proof for $\cdot\mathbf{;}\cdot$ requires many case distinctions and is not very enlightening. We will only prove one case here as an example.

Lemma 1. *Let $t_1 : A \rightleftarrows B$ be reverse semi-inverse and $t_2 : B \rightleftarrows C$ be pseudoinverse. Then $t_1\mathbf{;}t_2 : A \rightleftarrows C$ is pseudoinverse.*

Proof. First, we have to prove $\overrightarrow{(t_1\mathbf{;}t_2)};\overleftarrow{(t_1\mathbf{;}t_2)};\overrightarrow{(t_1\mathbf{;}t_2)} \sqsubseteq \overrightarrow{(t_1\mathbf{;}t_2)}$. The left-hand side simplifies by definition 1 to

$$\overrightarrow{(t_1\mathbf{;}t_2)};\overleftarrow{(t_1\mathbf{;}t_2)};\overrightarrow{(t_1\mathbf{;}t_2)} = \overrightarrow{t_1};\overrightarrow{t_2};\overleftarrow{t_2};\overleftarrow{t_1};\overrightarrow{t_1};\overrightarrow{t_2}\ .$$

Since $\overleftarrow{t_1};\overrightarrow{t_1} \sqsubseteq \mathrm{id}$ by assumption, and $\overrightarrow{t_2}$ is continuous and thus monotone, we have

$$\overrightarrow{(t_1\mathbf{;}t_2)};\overleftarrow{(t_1\mathbf{;}t_2)};\overrightarrow{(t_1\mathbf{;}t_2)} \sqsubseteq \overrightarrow{t_1};\overrightarrow{t_2};\overleftarrow{t_2};\overrightarrow{t_2}\ .$$

We also have $\overrightarrow{t_2};\overleftarrow{t_2};\overrightarrow{t_2} \sqsubseteq \overrightarrow{t_2}$ by assumption and can conclude

$$\overrightarrow{(t_1\mathbf{;}t_2)};\overleftarrow{(t_1\mathbf{;}t_2)};\overrightarrow{(t_1\mathbf{;}t_2)} \sqsubseteq \overrightarrow{t_1};\overrightarrow{t_2} = \overrightarrow{(t_1\mathbf{;}t_2)}\ .$$

$\overleftarrow{(t_1\mathbf{;}t_2)};\overrightarrow{(t_1\mathbf{;}t_2)};\overleftarrow{(t_1\mathbf{;}t_2)} \sqsubseteq \overleftarrow{(t_1\mathbf{;}t_2)}$ is proven analogously, but using the assumption $\overleftarrow{t_2};\overrightarrow{t_2};\overleftarrow{t_2} \sqsubseteq \overleftarrow{t_2}$. Thus, $t_1\mathbf{;}t_2$ is pseudoinverse. □

Figure 2c shows how janus classes are included in each other. This complete lattice defines a partial order, $\leq$, that we can use to broaden a janus type.

Finally, we look at how janus classes compose in parallel. We can prove in general that they are closed under any bifunctor.

Definition 4. *A bifunctor F is a mapping from pairs of types to types and also a (continuous) mapping from pairs of functions to functions, such that*

1. $f : A \to B \wedge g : C \to D \implies F\,(f, g) : F\,(A, C) \to F\,(B, D)$,
2. $F\,(\mathrm{id}_A, \mathrm{id}_B) = \mathrm{id}_{F\,(A,B)}$, *and*
3. $F\,(f_1;f_2, g_1;g_2) = (F\,(f_1, g_1));(F\,(f_2, g_2))$.

We extend a bifunctor F to januses, such that for any two januses $t_1 : A \rightleftarrows B$ and $t_2 : C \rightleftarrows D$, $F\,(t_1, t_2) = (F\,(\overrightarrow{t_1}, \overrightarrow{t_2}), F\,(\overleftarrow{t_1}, \overleftarrow{t_2}))$. Thus, we have $F\,(\mathbf{id}, \mathbf{id}) = \mathbf{id}$, $F\,(t_1\mathbf{;}t_2, t_3\mathbf{;}t_4) = (F\,(t_1, t_3))\mathbf{;}(F\,(t_2, t_4))$ and $F\,(t_1^{\dagger}, t_2^{\dagger}) = (F\,(t_1, t_2))^{\dagger}$ from the functor laws.

Remark 3. The parallel composition, $\otimes$, from remark 2 is actually a bifunctor extended to januses. The bifunctor maps a pair of functions, (f, g), to the function $\lambda(a, b).(f\,a, g\,b)$.

Again, we shall show proof for only one janus class as an example.

Lemma 2. *Let $t_1 : A \rightleftarrows B$ and $t_2 : C \rightleftarrows D$ be both semi-inverse and F be a bifunctor, then $F\,(t_1, t_2) : F\,(A, C) \rightleftarrows F\,(B, D)$ is also semi-inverse.*

Proof. We have to prove $\overrightarrow{(F\,(t_1,t_2))};\overleftarrow{(F\,(t_1,t_2))} \sqsubseteq \mathrm{id}$. The left-hand side simplifies as follows

$$\overrightarrow{(F\,(t_1,t_2))};\overleftarrow{(F\,(t_1,t_2))} = F\,(\overrightarrow{t_1},\overrightarrow{t_2});F\,(\overleftarrow{t_1},\overleftarrow{t_2}) = F\,(\overrightarrow{t_1};\overleftarrow{t_1},\overrightarrow{t_2};\overleftarrow{t_2})$$

by definition 4. Because F is monotone and $\overrightarrow{t_i};\overleftarrow{t_i} \sqsubseteq \mathrm{id}$ for $i \in \{1,2\}$, we have

$$\overrightarrow{(F\,(t_1,t_2))};\overleftarrow{(F\,(t_1,t_2))} \sqsubseteq F\,(\mathrm{id},\mathrm{id}) = \mathrm{id}\ .$$

Thus, $F\,(t_1,t_2)$ is semi-inverse. □

2.2 Choice

With $\otimes$ it is possible to compose reversible functions in parallel. We now define the functor $\oplus$ that composes januses as alternatives of each other.

Definition 5. *Let $\oplus$ be the bifunctor with*

$$\begin{aligned} A \oplus B &= A + B \\ t_1 \oplus t_2 &= \lambda x \mid \begin{cases} x = \mathrm{inj}_1\,a \bullet \mathrm{inj}_1(\overrightarrow{t_1}\,a) \\ x = \mathrm{inj}_2\,b \bullet \mathrm{inj}_2(\overrightarrow{t_2}\,b) \end{cases} \end{aligned}$$

where $A + B$ is the sum type, and $\mathrm{inj}_1 : A \to A + B$ and $\mathrm{inj}_2 : B \to A + B$ are the corresponding injections.

To actually make the choice, we use the following janus constructor.

Definition 6. *Let $\mathbf{if}_{\mathrm{si}} : (A \to \mathrm{Bool}) \to (A \rightleftarrows A + A)$ with*

$$\mathbf{if}_{\mathrm{si}}\,c = \begin{pmatrix} \lambda x \mid \begin{cases} \ \ c\,x \bullet \mathrm{inj}_1\,x \\ \neg c\,x \bullet \mathrm{inj}_2\,x \end{cases} \\ \lambda y \mid \begin{cases} y = \mathrm{inj}_1\,a \bullet a \\ y = \mathrm{inj}_2\,a \bullet a \end{cases} \end{pmatrix} .$$

Since there is an isomorphism between $A + A$ and $A \times \mathrm{Bool}$, $\mathbf{if}_{\mathrm{si}}$ is just a special form of $\mathbf{forget}^{\dagger}$. Thus, $\mathbf{if}_{\mathrm{si}}$ is not inverse, but only semi-inverse. It discards one bit of information in the reverse direction. Therefore, we cannot use it to construct inverse, reverse semi-inverse, or pseudoinverse januses. We define an inverse version that checks whether the correct alternative was used in the reverse direction.

Definition 7. *Let $\mathbf{if}_{\mathrm{in}} : (A \to \mathrm{Bool}) \to (A \rightleftarrows A + A)$ with*

$$\mathbf{if}_{\mathrm{in}}\,c = \begin{pmatrix} \lambda x \mid \begin{cases} \ \ c\,x \bullet \mathrm{inj}_1\,x \\ \neg c\,x \bullet \mathrm{inj}_2\,x \end{cases} \\ \lambda y \mid \begin{cases} y = \mathrm{inj}_1\,a \wedge \ \ c\,a \bullet a \\ y = \mathrm{inj}_2\,a \wedge \neg c\,a \bullet a \end{cases} \end{pmatrix} .$$

With these janus constructors as building blocks, it is possible to define a janus from alternative januses depending on a condition and an assertion. For example, the janus $\mathbf{if}_{\mathrm{si}}\,c\mathbf{;}\,(t \oplus e)\,\mathbf{;}(\mathbf{if}_{\mathrm{in}}\,a)^{\dagger}$ is semi-inverse if t and e are semi-inverse. For the compound janus to be defined at all, c and a should be related predicates. One typically expresses the same condition as the other, but in terms of a different data type.

2.3 Higher-Order Januses

Since we want to create a reversible language that has as many features of irreversible languages as possible, we have to look at januses over januses. The first non-trivial higher-order janus that comes to mind, is the janus that reverses other januses.

Definition 8. *Let* $\mathbf{rev}_{A,B} : (A \rightleftarrows B) \rightleftarrows (B \rightleftarrows A)$ *with* $\mathbf{rev}_{A,B} = (\cdot^{\dagger}, \cdot^{\dagger})$.

rev is a bijection and thus inverse.

A core concept of functional programming is the existence of a function $\mathrm{eval} : \mathrm{A} \times (\mathrm{A} \to \mathrm{B}) \to \mathrm{B}$, that applies a function to a value and returns the result. Turning this signature into a janus, like $\mathbf{eval}' : A \times (A \rightleftarrows B) \rightleftarrows B$, is not going to work. How are we supposed to compute from just a value the function and its argument from which the value originated? Instead, we use a trick. If not only the result of the janus application is returned, but also the janus, then we can define a useful, higher-order janus.

Definition 9. *Let* $\mathbf{jeval}_{A,B} : A \times (A \rightleftarrows B) \rightleftarrows B \times (A \rightleftarrows B)$ *with*

$$\overrightarrow{\mathbf{jeval}_{A,B}}\,(a,t) = (\overrightarrow{t}\;a, t)$$
$$\overleftarrow{\mathbf{jeval}_{A,B}}\,(b,t) = (\overleftarrow{t}\;b, t)\ .$$

Currying is another core concept of functional programming. Since currying is a bijection, we can define it as a janus.

Definition 10. *Let* $\mathbf{curry}_{A,B,C} : (C \to A \to B) \rightleftarrows (A \times C \to B)$ *with*

$$\overrightarrow{\mathbf{curry}_{A,B,C}}\,f = \lambda(a,c) \bullet f\,c\,a$$
$$\overleftarrow{\mathbf{curry}_{A,B,C}}\,f = \lambda c \bullet \lambda a \bullet f\,(a,c)\ .$$

Again, there is no obvious correspondent in the janus world. Using a similar trick as above we can find the following janus.

Definition 11. *Let* $\mathbf{jcurry}_{A,B,C} : (C \to (A \rightleftarrows B)) \rightleftarrows (A \times C \rightleftarrows B \times C)$ *with*

$$\overrightarrow{\mathbf{jcurry}_{A,B,C}}\,f = (\lambda(a,c) \bullet (\overrightarrow{(f\,c)}\,a, c), \lambda(b,c) \bullet (\overleftarrow{(f\,c)}\,b, c)$$
$$\overleftarrow{\mathbf{jcurry}_{A,B,C}}\,t = \lambda c \bullet (\lambda a \bullet \pi_1\,(\overrightarrow{t}\;(a,c)), \lambda b \bullet \pi_1\,(\overleftarrow{t}\;(b,c)))\ .$$

The argument of type C acts like a context in which the transformation between A and B is performed (explaining our unusual choice of type variable names).

Remark 4. An interesting fact is that **jeval** can be derived from **jcurry** by $\mathbf{jeval}_{A,B} = \overrightarrow{\mathbf{jcurry}_{A,B,A\rightleftarrows B}}\ \mathrm{id}_{A\rightleftarrows B}$.

$\overrightarrow{\textbf{jcurry}}$ suggests an idea how to bridge the irreversible and reversible world. A function $f : C \to (A \rightleftarrows B)$ could be any *irreversible* function that computes a janus. $\overrightarrow{\textbf{jcurry}}$ turns this into a janus, that we can use as a building block to compose more complex januses from. One variable of type A (or more than one, if A is a tuple) is consumed to produce a new variable of type B, while using – but not consuming – a variable of type C in that transformation. What this also suggests is that, when some expression is applied to a janus, the janus itself may be computed in an irreversible fashion from all variables that are not consumed in that expression.

2.4 Recursion

Since januses are just pairs of functions, defining a generic janus recursively by taking the fixpoint, fix F, of a function $F : (A \rightleftarrows B) \to (A \rightleftarrows B)$ just works. For recursion to make sense for other janus classes, the janus class has to be ω-complete and contain $\bot$. In this case, we can apply fixpoint induction. Again, we prove this for semi-inverse januses as an example.

Lemma 3. *Let $F : (A \rightleftarrows B) \to (A \rightleftarrows B)$ preserve semi-inverse januses. Then* fix F *is semi-inverse.*

Proof. By fixpoint induction.

1. Since $\overrightarrow{\bot};\overleftarrow{\bot} = \bot \sqsubseteq \mathrm{id}$, $\bot$ is semi-inverse.
2. F preserves semi-inverse januses by assumption.
3. Let $t_1 \sqsubseteq t_2 \sqsubseteq \ldots$ be an ω-chain of semi-inverse januses, and let $t = \bigsqcup_i t_i$. Then we have

$$\overrightarrow{t};\overleftarrow{t} = \left(\bigsqcup_{i\in\omega} \overrightarrow{t_i}\right);\left(\bigsqcup_{i\in\omega} \overleftarrow{t_i}\right) = \bigsqcup_{i_1\in\omega}\bigsqcup_{i_2\in\omega} \overrightarrow{t_{i_1}};\overleftarrow{t_{i_2}} = \bigsqcup_{i\in\omega} \overrightarrow{t_i};\overleftarrow{t_i} \sqsubseteq \bigsqcup_{i\in\omega} \mathrm{id} = \mathrm{id}\ .$$

Thus, t is semi-inverse and the set of semi-inverse januses is ω-complete. □

3 Generalized Reversible Functional Programming

In the previous chapter, we have effectively defined a point-free language for generalized reversible functional programming. This chapter will do the same in an applicative style. The syntax for this language is given in Fig. 3.

$$
\begin{array}{rcl}
J & ::= & \mathrm{in}|\mathrm{si}|\mathrm{rs}|\mathrm{pi}|\mathrm{gj}|\mathrm{ir} \\
E & ::= & V|\texttt{?}V \\
 & | & (E, \ldots, E) \\
 & | & E\,E \\
 & | & \lambda_J E \Rightarrow E\,|\,\ldots\,|\,E \Rightarrow E \\
 & | & S\,\texttt{;}\,E
\end{array}
\qquad
\begin{array}{rcl}
T & ::= & \mathrm{Top}|\mathrm{Bot}|\mathrm{Equ}|\mathrm{Bool}|\mathrm{Int}|\mathrm{List}| \\
 & | & T \times \cdots \times T \\
 & | & T \;{\scriptstyle\leftarrow}J{\scriptstyle\rightarrow}\; T \\
S & ::= & S\,\texttt{;}\,S \\
 & | & \texttt{let}\; E \Leftarrow E \\
 & | & \texttt{forget}\; E \Leftarrow E \\
 & | & \texttt{remember}\; E \Leftarrow E
\end{array}
$$

Fig. 3: Syntax of the reversible language

Not surprisingly we generalized function types $(A \rightarrow B)$ to janus types $(A \;{\scriptstyle\leftarrow}J{\scriptstyle\rightarrow}\; B)$ by including the janus class. We also distinguish types with equality and without equality. Those with equality are a subtype of Equ. As expected from the type signature $(A \rightarrow B) \times (B \rightarrow A)$, reversible januses are invariant in their type arguments. But they are covariant in their janus class, i.e., a janus class $A \;{\scriptstyle\leftarrow}J_1{\scriptstyle\rightarrow}\; B$ is considered a subclass of $A \;{\scriptstyle\leftarrow}J_2{\scriptstyle\rightarrow}\; B$ if and only if $J_1 \leq J_2$.

What probably is most unusual is the omission of a sub-language for patterns. In this language all expressions can be used as a pattern. A definition involving a function application, $\texttt{let}\; x = f\,e$ for example, can be easily reversed if f is a reversible function. The reverse is $\texttt{let}\; e = f^{\dagger}\,x$. This is equivalent to writing $\texttt{let}\; f\,e = x$ in our language. Thus, when the term $f\,e$ is used to pattern match a value v, the reverse direction of the value of f is applied to v and this transformed value is then pattern matched against e.

The only exception is that λ-constructs cannot be used as a pattern. Matching against a λ-construct would mean finding the values of the free variables in the λ-construct that make it equal to the function matched against. This is undecidable in general. Thus, λ-expressions have to be restricted to the body of irreversible functions. Here, we do not enforce this restriction in the syntax to simplify the denotational semantics.

The denotational semantics, $[\![\cdot]\!]_E$, in Fig. 4 assigns each expression a janus of type $\Gamma \rightleftarrows \mathrm{Top} \times \Gamma$, where $\Gamma = V \rightarrow \mathrm{Top}$ is the type of variable assignments. The normal direction defines the semantics when the expression is used as a value. The reverse direction defines the semantics when the expression is used as a pattern.

Let-expressions are generalized to statements and the scoping expression, $s\,\texttt{;}\,e$. The `forget`- and `remember`-statements allow the explicit discarding and reconstruction of information. A statement's denotation, $[\![\cdot]\!]_S : \Gamma \rightleftarrows \Gamma$, is simply a janus between environments, as it can only produce and consume variables.

Tuple expressions evaluate their components from left to right. Thus, values consumed in the right sub-expression are available to the left sub-expression, but not vice versa. Pattern matching of tuples happens necessarily in the opposite direction, from right to left.

$$
\begin{aligned}
\llbracket v \rrbracket_E &= \begin{pmatrix} \lambda\gamma \bullet (\gamma\, v, \gamma) \\ \lambda(x,\gamma) \mid \gamma\, v == x \bullet \gamma \end{pmatrix} \\
\llbracket \mathtt{?}v \rrbracket_E &= \begin{pmatrix} \lambda\gamma \bullet (\gamma\, v, \gamma \backslash v) \\ \lambda(x,\gamma) | v \notin \mathrm{dom} \bullet \gamma \cup (v \mapsto x) \end{pmatrix} \\
\llbracket (e_1, \ldots, e_n) \rrbracket_E &= \llbracket e_1 \rrbracket \mathbin{\hat{\otimes}} (\ldots \mathbin{\hat{\otimes}} \llbracket e_n \rrbracket) \\
& \text{where } a \hat{\otimes} b = a\mathbf{;}(\mathbf{id} \otimes b)\mathbf{;assoc}^{\dagger} \\
\llbracket f\, e \rrbracket_E &= \llbracket e \rrbracket_E \,\mathbf{;}\overrightarrow{\mathbf{jcurry}}\, (\overrightarrow{\llbracket f \rrbracket_E ;\pi_1}) \\
\llbracket \lambda_J p_1 \Rightarrow e_1 \mid \ldots \mid p_n \Rightarrow e_n \rrbracket_E &= \begin{pmatrix} \lambda\gamma \bullet \left((\mathbf{forget}\, \lambda x.\gamma)^{\dagger}\mathbf{;}\beta\mathbf{;forget}\, \lambda y.\gamma, \gamma\right) \\ - \end{pmatrix} \\
& \text{where } \beta = \llbracket p_1 \rrbracket_E^{\dagger} \,\mathbf{;}\, \llbracket e_1 \rrbracket_E \mathbin{\hat{\oplus}^J} (\ldots \mathbin{\hat{\oplus}^J} \llbracket p_n \rrbracket_E^{\dagger} \,\mathbf{;}\, \llbracket e_n \rrbracket_E) \\
& \text{where } a \mathbin{\hat{\oplus}^J} b = \mathbf{if}_J\, (\overrightarrow{a} \cdot \neq \mathfrak{U})\mathbf{;}\, (a \oplus b)\, \mathbf{;}(\mathbf{if}_{J^\dagger}\, (\overleftarrow{a} \cdot \neq \mathfrak{U}))^{\dagger} \\
& \text{where } \mathbf{if}_{\mathrm{in}} = \mathbf{if}_{\mathrm{rs}} = \mathbf{if}_{\mathrm{pi}} \text{ and } \mathbf{if}_{\mathrm{si}} = \mathbf{if}_{\mathrm{gj}} = \mathbf{if}_{\mathrm{ir}} = \mathbf{if}_{\mathrm{ir}^\dagger} \\
\llbracket s \mathbin{;} e \rrbracket_E &= \llbracket s \rrbracket_S \mathbin{;} \llbracket e \rrbracket_E \\
\llbracket s_1 \mathbin{;} s_2 \rrbracket_S &= \llbracket s_1 \rrbracket_S \mathbin{;} \llbracket s_2 \rrbracket_S \\
\llbracket \mathtt{let}\, p \Leftarrow e \rrbracket_S &= \llbracket e \rrbracket_E \mathbin{;} \llbracket p \rrbracket_E^{\dagger} \\
\llbracket \mathtt{forget}\, p \Leftarrow e \rrbracket_S &= \llbracket p \rrbracket_E \,\mathbf{;swap;forget}\, (\overrightarrow{\llbracket e \rrbracket_E ;\pi_1}) \\
\llbracket \mathtt{remember}\, p \Leftarrow e \rrbracket_S &= \llbracket \mathtt{forget}\, p \Leftarrow e \rrbracket_S^{\dagger}
\end{aligned}
$$

Fig. 4: Denotational semantics of the reversible language

From the discussions about **dup** and **jcurry** we derive the following rules regarding variables:

1. Variables of types with equality may be used multiple times. They act as duplicates, when used as a value, or equality checks, when used as a pattern. This allows us to omit literal values in the syntax. Literal values can be emulated by defining them as variables in the outermost scope.
2. If the current scope is that of an irreversible function, variable usage is loosened to the normal define-before-use rule. There, even types without equality may be duplicated.
3. The janus sub-expression of a janus application is an irreversible scope and has access to all variables that are not consumed or produced in the argument sub-expression. The same is true for the right sub-expression of a `forget`- and `remember`-statement.
4. Otherwise, variables must be defined exactly once and then consumed exactly once. In order to differentiate between a definition/consumption use and a copy/equality test use of a variable, we introduce the `?`V form. This is only necessary in order to keep the semantics simple and compositional. Every variable must be defined using the `?`V form. In a reversible scope, the last usage of every variable must also be a `?`V form.

The other peculiarity of these semantics is how we treat pattern matching. Januses defined in this system are implicitly using the Maybe (or Option) monad. The special semantic value $\mathfrak{U}$ is used to denote when a janus is undefined at the given argument, the None (or Nothing) case. This includes the conditional lambda expression, that we have used until now, i.e., $\neg P(x) \implies (\lambda y \mid P(y) \bullet E(y))\, x = \mathfrak{U}$. All functions are implicitly strict in $\mathfrak{U}$, i.e., $f\,\mathfrak{U} = \mathfrak{U}$. The lambda expression is the only place where we treat $\mathfrak{U}$ in a special way and this is where the pattern matching happens. Also, note that $\mathfrak{U}$ is different from $\perp$. We leave $\perp$ as the semantic value for non-termination. Especially, non-termination in a predicate still leads to a non-terminating function, i.e., $P(x) = \perp \implies (\lambda y \mid P(y) \bullet E(y))\, x = \perp$.

Remark 5. This treatement is similar to exceptions in the programming language Haskell [3]. $\mathfrak{U}$ can be emulated by a special exception and pattern matching is then just syntactic sugar for handling this exception.

Pattern matching generally happens in a similar way to other functional languages. The sub-cases of a function are tried in order. The first case that matches, i.e., is not $\mathfrak{U}$, is the one that determines the output. But from the discussions about $\mathbf{if}_{\mathrm{si}}$ and $\mathbf{if}_{\mathrm{in}}$, we know that, depending on the janus class, we have to perform some consistency checking afterwards. For inverse, reverse semi-inverse, and pseudoinverse januses, we have to ensure that none of the cases, that were undefined in the normal direction, are defined in the reverse direction. The same is true when going backwards for inverse, semi-inverse, and pseudoinverse januses.

3.1 An Example

As an example, we will define the janus $parseInt : \mathrm{List} \leftarrow\!\mathrm{rs}\!\rightarrow \mathrm{Int}$ that computes from a list of digits the corresponding number and vice versa. $parseInt$ shall discard leading zeros and, therefore, is reverse semi-inverse. For this example, we assume the constants true : Bool, false : Bool, nil : List and cons : $\mathrm{Int} \times \mathrm{List} \leftarrow\!\mathrm{in}\!\rightarrow \mathrm{List}$ are already defined in the global context with their usual meaning. These are usually defined as type constructors in irreversible languages. Since type constructors are always injective, they extend naturally to januses. We assume the binary operators $+$ and $*$ of type $\mathrm{Int} \leftarrow\!\mathrm{ir}\!\rightarrow (\mathrm{Int} \leftarrow\!\mathrm{in}\!\rightarrow \mathrm{Int})$ which perform addition/subtraction and multiplication/partially defined division respectively. The syntactic sugar for these operators swaps the arguments, i.e., $l + r \equiv (\cdot + \cdot)\, r\, l$. Hence, it is the left operand which is consumed, and the right operand stays untouched. We also assume $// : \mathrm{Int} \leftarrow\!\mathrm{ir}\!\rightarrow (\mathrm{Int} \leftarrow\!\mathrm{ir}\!\rightarrow \mathrm{Int})$ which performs integer division with rounding towards negative infinity.

The janus $d2n$ in Fig. 5 is a first step. It uses the janus constructor muladd : $\mathrm{Int} \leftarrow\!\mathrm{ir}\!\rightarrow (\mathrm{Int} \times \mathrm{Int} \leftarrow\!\mathrm{rs}\!\rightarrow \mathrm{Int})$ with the following behavior:

$$\mathrm{muladd}\, k = \begin{pmatrix} \lambda(a, b) \bullet a * k + b \\ \lambda y \bullet (\lfloor y/k \rfloor, y \bmod k) \end{pmatrix} .$$

```
let muladd ⇐ λir ?k ⇒
  λrs (?a, ?b) ⇒
    let ?w ⇐ ?a * k;
    let ?y ⇐ ?b + w;
    forget ?w ⇐ y // k * k;
    ?y;
let d2n ⇐ λrs
    nil ⇒ 0
  | cons(?d, ?l) ⇒
      muladd 10 (d2n ?l, ?d);
let append ⇐ λin
    (nil, ?x) ⇒ cons(?x, nil)
  | (cons(?y, ?l), ?x) ⇒
      cons(?y, append(?l, ?x));
let reverse ⇐ λin
    nil ⇒ nil
  | cons(?x, ?l) ⇒
      append(reverse ?l, ?x);
let compose_rs ⇐ λir ?f ⇒ λir
  ?g ⇒ λrs ?x ⇒ g (f ?x);
let parseInt ⇐
  compose_rs reverse d2n;
```

```
let divmod ⇐ λir ?k ⇒
  λsi ?y ⇒
    remember ?w ⇐ y // k * k;
    let ?b ⇐ ?y − w;
    let ?a ⇐ ?w / k;
    (?a, ?b);
let n2d ⇐ λsi
    0 ⇒ nil
  | ?x ⇒ let (?n, ?d) ⇐ divmod 10 ?x;
      cons (?d, n2d ?n);
let unappend ⇐ λin
    cons(?x, nil) ⇒ (nil, ?x)
  | cons(?y, unappend† (?l, ?x)) ⇒
      (cons(?y, ?l), ?x);
let unreverse ⇐ λin
    nil ⇒ nil
  | ?y ⇒ let (?r, ?x) ⇐ unappend ?y;
      cons (?x, unreverse ?r);
let compose_si ⇐ λir ?f ⇒ λir
  ?g ⇒ λsi ?x ⇒ g (f ?x);
let prettyPrintInt ⇐
  compose_si n2d unreverse;
```

Fig. 5: Semi-reversible number parser and its reverse

The statement `let` $?y \Leftarrow ?b+w$ is where we use $\overrightarrow{\textbf{jcurry}}$ to its full extent. The variable w is used here in an irreversible context, even though it is defined in a reversible context. This is sound, because w does not appear in the argument to the janus $\cdot + w$, which is just b.

The implementation of *d2n* is straight-forward, but it has a problem: It reads the digits in the wrong order. We want the most significant digit to be the first in the list. This is solved by the janus *reverse*, which is written in terms of the janus *append*. Their definition is no different from that in an irreversible, functional language. The definition just happens to be reversible.

Finally, we define *parseInt* in a point-free style, to show-case this possibility.

Figure 5 also shows an implementation of the reverse for each of the above januses and janus constructors. The reverse of any lambda expression, $\lambda_J p \Rightarrow b$, is simply $\lambda_{J^\dagger} e \Rightarrow p$. This does not help much in understanding what is going on. From the denotational semantics (Fig. 4) we can derive the simplification rules in Fig. 6. The implementations utilizes those simplifications, in order to make the source code easier to understand.

$$\begin{aligned}
[\![\texttt{let}\ e_1\, e_2 \Leftarrow e_3]\!]_S &= \left[\!\!\left[\texttt{let}\ e_2 \Leftarrow e_1^\dagger\, e_3\right]\!\!\right]_S \\
[\![\lambda_J \texttt{let}\ e_1 \Leftarrow e_2; e_3 \Rightarrow e_4]\!]_E &= [\![\lambda_J e_3 \Rightarrow \texttt{let}\ e_2 \Leftarrow e_1; e_4]\!]_E \\
[\![\lambda_J e_1\, e_2 \Rightarrow e_3]\!]_E &= [\![\lambda_J e_1\, ?x \Rightarrow \texttt{let}\ e_2 \Leftarrow ?x; e_2]\!]_E \quad \text{with } x \text{ new variable} \\
[\![\lambda_J \texttt{forget}\ e_1 \Leftarrow e_2; e_3 \Rightarrow e_4]\!]_E &= [\![\lambda_J e_3 \Rightarrow \texttt{remember}\ e_1 \Leftarrow e_2; e_4]\!]_E
\end{aligned}$$

Fig. 6: Useful equivalences

4 Related Work

The observation that duplication in one direction requires equality testing in the other direction – as in **dup** – has been noted before [5,10].

The construction $\mathbf{if}_{\text{in}}\, c\mathbf{;}\, (t \oplus e)\,\mathbf{;}(\mathbf{if}_{\text{in}}\, a)^{\dagger}$ is inspired and closely related to the `if`-statement in the programming language Janus. Our pattern matching algorithm is an instance of that construction. It actually generalizes the symmetric first-match policy from RFun.

Our treatment of januses as irreversible expressions – as motivated by $\overrightarrow{\mathbf{jcurry}}$ – is a generalization of reversible updates from [9].

Lenses are isomorphic to a subset of inverse januses. A lens l consists of two functions $l.get : A \to B$ and $l.put : B \times A \to A$ adhering to the lens laws

$$\begin{aligned} l.get(l.put(b,a)) &\sqsubseteq b \\ l.put(l.get(a),a) &\sqsubseteq a \ . \end{aligned}$$

These laws are equivalent to requiring the janus $t : A \rightleftarrows B \times A$ to be inverse and have $\overrightarrow{t};\pi_2 = \mathrm{id}_A$.

5 Future Work

A more thorough domain theoretic treatment of the reversible language is needed. Issues like recursion and loops will then have a stronger basis.

A proper type system should prove that defined januses actually belong to their respective janus class. The janus class of a reversible λ-expression is almost derivable from the denotational semantics from Fig. 4 using the foundations that were laid out in Section 2. We must additionally prove that $[\![v]\!]_E$ and $[\![?v]\!]_E$ are inverse, and that $\overrightarrow{\mathbf{jcurry}}\,(\overrightarrow{[\![f]\!]_E};\pi_1)$ has the same janus class as $\overrightarrow{[\![f]\!]_E}$.

A category theoretic treatment might also be enlightening. Especially in regards to quantum computation, which shares many properties with reversible programming and in recent years got their share of category theory in [7].

The same holds for linear type systems. The language proposed here has a lot in common with linear languages, only we do not consume variables in the function part of a function application. This suggests a modification of the modus ponens in linear logic to $\dfrac{A \to B \quad A}{A \to B \quad B}$, where the implication may be reused.

A usability issue comes up when we define binary operators like $\cdot + \cdot : \mathrm{Int} \to (\mathrm{Int} \rightleftarrows \mathrm{Int})$. Only one of its arguments can be consumed. The other one must be constant in the current scope. But which one? Is it necessary to have two versions of each binary operator? Is it useful to have typing rules that can deal with overloaded operators? Or is our current approach, where the left operand is always consumed, sufficient?

6 Introduction†

We have defined januses as the core concept of a reversible program along with other concepts to compose large januses from smaller ones. The functors $\otimes$ and $\oplus$ for parallel and alternative execution, **if** as basis for choice and pattern matching, **forget** as means to discard information and **jcurry** as the bridge between the irreversible and reversible world. One fundamental observation is that, unlike injective functions, general januses *can* discard information, but unlike irreversible functions, they cannot discard information *implicitly*.

We have also given the syntax and semantics for a programming language that allows to write reversible and irreversible januses. Recursion, pattern matching, and higher order programming are possible and even look like functional programming. The language allows to use irreversible functions to compute reversible januses and invoke them in even in a reversible context. As a side effect, this language generalizes pattern matching to all expressions except lambda abstractions. It especially allows to pattern-match against a janus application.

Januses with certain consistency requirements have been identified. These janus classes compose with relatively simple rules and form a complete lattice. Our example program shows how a very simplified parser can be written as a reverse semi-inverse janus. Its reverse, the pretty-printer, is implicitly specified by the same source code.

References

1. Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: Resourceful Lenses for String Data. In *ACM SIGPLAN Notices*, pages 407–419, 2008.
2. Nate Foster, Kazutaka Matsuda, and Janis Voigtländer. Three Complementary Approaches to Bidirectional Programming. *Generic and Indexed Programming*, pages 1–46, 2012.
3. Simon Peyton Jones, Alastair Reid, Tony Hoare, and Fergus Henderson. A Semantics for Imprecise Exceptions. In *ACM SIGPLAN Notices*, pages 25–36, 1999.
4. Peter Kourzanov. Bidirectional Parsing – A Functional/Logic Perspective. *International Symposia on Implementation and Application of Functional Languages (IFL 2014)*, 2014.
5. Shin-cheng Mu, Zhenjiang Hu, and Masato Takeichi. An Algebraic Approach to Bi-directional Updating. In *Programming Languages and Systems*, pages 2–20, 2004.
6. Tillmann Rendel and Klaus Ostermann. Invertible Syntax Descriptions: Unifying Parsing and Pretty Printing. In *ACM Haskell Symposium*, pages 1–12, 2010.
7. Peter Selinger. Dagger compact closed categories and completely positive maps (extended abstract). pages 1–23, 2005.
8. John Michael Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International, 2nd edition, 1998.
9. Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. Principles of a Reversible Programming Language. In *Computing Frontiers*, pages 43–54, 2008.
10. Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. Towards a Reversible Functional Language. In *Reversible Computation*, pages 14–29, 2012.

Exchange between the German and French Compiler Communities

Florian Brandner[1] and Jens Knoop[2]

[1] Département Informatique et Réseaux (INFRES)
Télécom ParisTech
Université Paris-Saclay
florian.brandner@telecom-paristech.fr

[2] Institut für Computersprachen
Fakultät für Informatik
Technische Universität Wien
knoop@complang.tuwien.ac.at

Extended Abstract It is becoming more and more common that research communities related to a specific topic organize themselves in order to gain visibility, foster collaboration, and coordinate the research activities among the community members. The resulting *organizations* are usually rather loosely structured, informal, and open to all researchers interested in participating.

In the field of programing languages, program analysis, and compilers the German-speaking research community has a long tradition of informal meetings that bring researchers and practitioners together. These meetings serve as an open forum for discussions, for spreading new ideas, but also help to meet other researchers, to foster exchange, and to develop new collaborations. Participants in those meetings easily can see how active and vibrant the German-speaking community in this field is – this is again proven by the high number of participants to this edition of KPS.

The French community is equally active and striving, and also has a, not equally long, but equally successful tradition in gathering its community members at informal meetings. Participants in events of both communities quickly realize that the French and the German meetings share a common spirit: fostering exchange.

Despite the success on both sides, the French- and the German-speaking communities as such stay largely separate – even-though visitors from abroad are in principle welcome. In this presentation, we will explain the current way that both *compiler* communities are organized and summarize the communities' activities. The ultimate goal is to gather feedback from the community members regarding the development of joint activities – for instance, joint meetings – in order to foster the exchange across boarders. These joint activities could even lead to the creation of a joint organizational structure such as an *international Groupement de Recherche*.[3]

[3] http://www.cnrs.fr/en/workingwith/GDRI.htm

Background and Links

Florian Brandner – together with Laure Gonnord and Fabrice Rastello – is co-coordinator of the French compilation community:

- Website of the Compilation group:
  http://compilation.gforge.inria.fr/
- The parent organization of the group:
  http://gdr-gpl.cnrs.fr/

Jens Knoop is elected chairman of the executive board of the special interest group on "Programmiersprachen und Rechenkonzepte":

- Website of the special interest group "Programmiersprachen und Rechenkonzepte":
  http://www-ps.informatik.uni-kiel.de/fg214/
- The parent organization of the special interest group:
  http://fb-swt.gi.de/softwaretechnik/

Acknowledgments This initiative is supported by a grant from the French Compilation Community and the Groupement de Recherche Génie de la Programmation et du Logiciel (GDR GPL).

Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity

Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz

University of California, Irvine

Abstract We explore software diversity as a defense against side-channel attacks by dynamically and systematically randomizing the control flow of programs. Existing software diversity techniques transform each program trace *identically*. Our diversity based technique instead transforms programs to make each program trace *unique*. This approach offers probabilistic protection against both online and off-line side-channel attacks. In particular, we create a large number of unique program execution paths by automatically generating diversified replicas for parts of an input program. Replicas derived from the same original program fragment have different implementations, but perform semantically equivalent computations. At runtime we then randomly and frequently switch between these replicas.

We evaluate how well our approach thwarts cache-based side-channel attacks, in which an attacker strives to recover cryptographic keys by analyzing side-effects of program execution. Our method requires no manual effort or hardware changes, has a reasonable performance impact, and reduces side-channel information leakage significantly.

Keywords: language-based security, software diversity, dynamic diversity, side channels.

[1]

1 Motivation

Artificial software diversity, like its biological counterpart, is a highly flexible and efficient defense mechanism. Code injection, code reuse, and reverse engineering attacks are all significantly harder against diversified software ([10,16,35,21,42,18,14,12]). We propose to extend software diversity to protect against side-channel attacks, in particular cache side channels.

Essentially, artificial software diversity denies attackers precise knowledge of their target by randomizing implementation features of a program. Because code reuse and other related attacks rely on *static properties* of a program, previous work on software diversity predominantly focuses on randomizing the program

[1] Note: This paper appeared at NDSS'15.

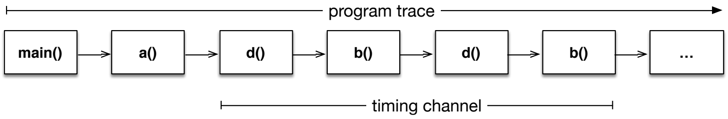


Figure 1: Time based side channel exploitable through a sequence of function calls in a program trace.

representation, e.g., the in-memory addresses of code and data. Side-channel attacks, on the other hand, rely on *dynamic properties* of programs, e.g., execution time, memory latencies, or power consumption. Consequently, diversification against side channels must randomize a program's execution rather than its representation.

Most existing diversification approaches randomize programs before execution, e.g., during compilation, installation, or loading. Ahead-of-time randomization is desirable because re-diversification during runtime impacts performance (similar to just-in-time compilation). Some approaches interleave program randomization and program execution ([30,18,38,22]). However, the granularity of randomization in these approaches is quite coarse, potentially allowing an attacker to observe the program uninterrupted for long enough to carry out a successful side-channel attack. We avoid this problem by extending techniques used to prevent reverse engineering such as code replication and control-flow randomization ([6,14]). Unlike these approaches, however, we replicate code at a finer grained level and produce a nearly unlimited number of runtime paths by randomly switching between these replicas. Rather than making control flow difficult to reverse engineer, our technique randomly switches execution between different copies of program fragments, which we refer to as replicas, to randomize executed code and thus side-channel observations. We call this new capability *dynamic control-flow diversity.*

To vary the side-channel characteristics of replicas, we employ diversifying transformations. Diversification preserves the original program semantics while ensuring that each replica differs at the level of machine instructions. To protect against cache side-channel attacks we use diversifications that vary observable execution characteristics. Like other cache side-channel mitigations, such as reloading the cache on context switches and rewriting encryption routines to avoid optimized lookup tables, introducing diversity has some performance impact which we rigorously quantify in this paper.

In combination, dynamic control-flow diversity and diversifying transformations create binaries with randomized program traces, without requiring hardware or developer assistance. In this paper we explore the use of dynamic control-flow diversity against cache-based side-channel attacks on cryptographic algorithms. Our main contributions are the following:

- We apply the new capability of dynamic control-flow diversity to the problem of side channels. To the best of our knowledge, this is the first use of automated software diversity to mitigate cache side channels.
- We show how to generate machine code for efficient randomized control-flow transfers and combine this with a diversifying transformation to counter cache-based side-channel attacks.
- We present a careful and detailed evaluation of applying diversity to protect cache side channels and report the following:
  - **Security**: Our techniques successfully mitigate two realistic cache side-channel attacks against AES on modern hardware.
  - **Performance**: Applying dynamic control-flow diversity with effective security settings to an AES micro-benchmark of the libgcrypt library results in performance impacts of 1.75x and protecting a real-world application using AES results in a slowdown of 1.5x.

2 Side-Channel Background

The execution of a program is described by its control flow. The sequence of all control-flow transitions a program takes during execution is usually referred to as an execution path, or a *program trace*. A program trace describes the dynamic behavior of a program. Figure 1 illustrates a program trace at the granularity of function calls.

Executing programs on real hardware results in dynamic properties that leak information, such as timing or power variation. For example, Figure 1 shows a side channel based on time spent in executing the function sequence d(), b(), d(), b(). By observing dynamic properties of a program trace through a side channel, attackers can derive information about the actual program execution, such as inferring secret inputs to the program.

2.1 Threat Model

Since side-channel attacks often target secret keys of a process performing encryption, in this paper we assume that an attacker is targeting such a secret key. To demonstrate the applicability of our techniques, we assume an advantageous scenario for this attacker and reason that our defense remains effective under weaker assumptions.

Tromer et al. [39] classified side-channel attacks into synchronous and asynchronous attacks depending on whether or not the attacker can trigger processing of known inputs (usually plain- or cipher-texts). Synchronous encryption attacks, where the attacker can trigger and observe encryption of known messages, are generally easier to perform, and thus harder to defend against, since the attack does not need to determine the start and end of each encryption. We assume as strong a position for the attacker as possible and therefore will consider the scenario where an attacker can request and observe encryption of arbitrary chosen plaintexts.

To minimize external noise, we assume that the attacker is co-resident on the same machine as the target process. We also assume that the attacker can execute arbitrary user-mode code on a processor core shared with the target process but does not have access to the address space of the target process.

In the interest of allowing a strong attacker model, we advise but do not require that the protected binary be kept secret. Since we randomly generate diverse but semantically equivalent binaries, preventing the attacker from reconstructing the target environment is an advisable defense-in-depth against off-line attacks, such as the cross-VM attack described by Zhang et al. [44]. Deploying protected programs with differing layouts is also an effective defense against code-reuse attacks [23] and we can defend in the same manner by deploying randomized binaries which include dynamic control-flow diversity.

If we allow access to the binary, we must be careful that the attacker is not able to accurately determine which replica of each program unit was executed in an observed program trace. An attacker who observes a complete trace of control-flow transfers could filter out the effects of the replicas' diversifying transformations, regardless of what those effects are. In practice, a user-level process cannot observe all control-flow transfers of another process, especially at the granularity of basic blocks[2].

2.2 Example Attacks

To demonstrate an example of our dynamic control-flow diversity defense, we chose two synchronous, known input cache attacks on AES described by Tromer et al. [39]: EVICT+TIME and PRIME+PROBE. Although these representative cache attacks have limited scope, an attacker could use this type of attack to compromise a system-wide filesystem encryption key or target a proxy server where an attacker can trigger encryption of known plaintexts. In addition, these attacks are representative of cache-based side channels and are the basis of several more complex side-channel attacks [44,43,36]. While we demonstrate the effectiveness of our technique against cache-based side channels in particular, we expect that the same general defense paradigm can be applied to other categories of side channels using different diversifying transformations than the ones we discuss in Section 3.1.

Caches exploit temporal and spatial locality to speed up access to recently used data. This helps to compensate for the speed gap between processors and main memories. As a side effect, caches increase the correlation between program inputs and its execution characteristics.

Modern processors access the cache in units called "cache lines," which are typically 64 bytes long. Each cache level is partitioned into n "cache sets," and each memory line can be placed into exactly one of these n sets. Each set stores at most m lines simultaneously, in which case the cache is called "m-way set

[2] Gullasch et al. [20] describe a DoS attack against the OS scheduler which could result in such fine-grained information, but the OS scheduler can be hardened to prevent such attacks.

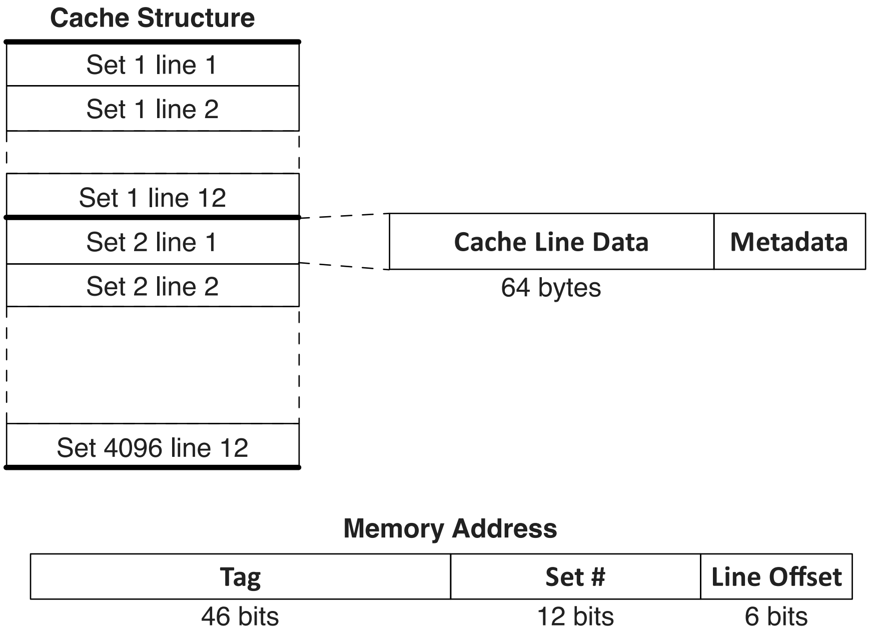


Figure 2: Example of cache structure on a modern processor. Cache shown is 3MB in size, with 4096 (2^{12}) sets, 12-way associativity and 64-byte cache lines. Memory addresses are broken into a 46-bit (or less) tag, a 12-bit set number and a 6-bit line offset.

associative". In practice, caches are 4-, 8-, 12- and 16-way associative. Figure 2 shows the structure of a 3MB 12-way set associative cache found in our test system.

For efficiency, the processor shares these caches between running processes but prevents processes from accessing data belonging to other processes via the virtual memory abstraction. However, since data from multiple processes is concurrently stored in the cache, adversaries can indirectly deduce information about which cache locations a target process accesses by observing side-effects of cache accesses. Since the data cache access patterns of many programs are input-dependent and predictable, attackers can use knowledge of some inputs and the target's data access patterns to derive the secret input.

To exploit cache access patterns, all cache timing attacks rely on the same fundamental principle of cache behavior: accessing data stored in the cache is measurably faster than accessing the data from main memory. As a result, attacks can exploit this principle as a side channel and observe different cache behavior for certain segments of a program trace. In the EVICT+TIME attack, we observe the effect of evicting an entire cache set and forcing the encryption program to

Algorithm 1: EVICT+TIME attack.

```
Input   : Cache set $c$ to probe, plaintext $p$, key $k$.
Output : Time needed to encrypt the plaintext after probing $c$.
Encrypt($k$, $p$);
Evict cache set $c$;
$t_0 \leftarrow$Time();
Encrypt($k$, $p$);
$t_1 \leftarrow$Time();
return $t_1 - t_0$;
```

Algorithm 2: PRIME+PROBE attack.

```
Input   : Array $C$ of cache sets to probe, plaintext $p$, key $k$.
Output : Array $T$ of times needed to probe each set in $C$.
foreach $c \in C$ do
    Read $w$ values into cache set $c$ from memory;
end
Encrypt($k$, $p$);
foreach $c \in C$ do
    $t_0 \leftarrow$Time();
    Read $w$ values from cache set $c$;
    $t_1 \leftarrow$Time();
    $T[c] \leftarrow t_1 - t_0$;
end
return $T$;
```

fetch values from main memory, while in the PRIME+PROBE attack we fill a cache set and check which cache lines the encryption evicts by observing the time to reload our data.

For convenience we summarize both AES attacks here but refer interested readers to Tromer et al. [39] for further details. Optimized AES implementations use four in-memory tables (T_0 through T_3, each containing 256 four-byte values) during encryption, and the access pattern of these tables varies according to the key and plaintext inputs. Specifically, during the first of ten encryption rounds for plaintext $\mathbf{p}$ and key k, the encryption process will access table T_l at index $p_i \oplus k_i$ for all $i = 0, \ldots, 15$ where $l = i \bmod 4$. Since we assume the attacker knows the plaintext $\mathbf{p}$, the attacker is able to derive information about the key from information about which table elements are loaded from memory.

Algorithm 1 shows the EVICT+TIME attack. We derive the table access patterns by observing the total execution time of the encryption routine. By first running the encryption on a chosen, random plaintext, we prime the cache with the table entries required during the encryption of this plaintext. We then completely evict a cache set by loading a set of memory locations that all map into the chosen cache set. By timing another encryption of the same plaintext, we can then, by averaging over many runs, determine whether the encryption

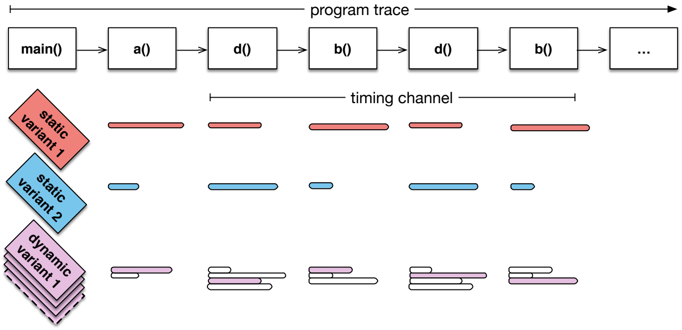


Figure 3: Side-channel resistance of diversification techniques.

used a table value from that cache set, since the encryption routine will take longer when accessing an evicted table entry due to the cache miss.

The PRIME+PROBE attack (shown in Algorithm 2) is very similar to the EVICT+TIME attack, but with the timing and eviction roles flipped. In this attack we first create a known starting cache state by loading a set of memory locations into each relevant cache set. We then trigger encryption of a chosen plaintext, which will modify this cache state by caching accessed table entries. Finally, we determine which cache sets were modified by timing a load of each cache set again. The cache sets corresponding to table entries that the encryption accessed will take longer to load than those not used, since the encryption table entry will have displaced one of the original entries loaded by the attacker and thus incur at least one cache miss.

By analyzing a large set of these cache observations for randomly chosen plaintexts, we can determine the key bits that correspond to table indices in the first round of encryption. For each guess of a key byte $\hat{k}_i$, we average all observed timings for the cache set evictions corresponding to table entry $T_{i \bmod 4}[\hat{k}_i \oplus p_i]$. In both attacks, the highest observed average time should correspond to the correctly guessed key byte. However, with 64 byte cache lines, four table entries fit into each cache line, and we can only observe accesses at the granularity of cache lines, which means that we can only determine the high nibble of each key byte with this analysis. Therefore, to determine the lower four bits of each key byte, we must analyze the second round of encryption as described by Tromer et al. This analysis, while more involved, is conceptually analogous to the first round analysis and we refer interested readers to the description in the original paper.

3 Dynamic Control-Flow Diversity

Most diversification techniques prevent attackers from constructing reliable attacks by randomizing the layout of a program's data and code. Since modern exploits such as code reuse attacks depend on detailed knowledge of the program

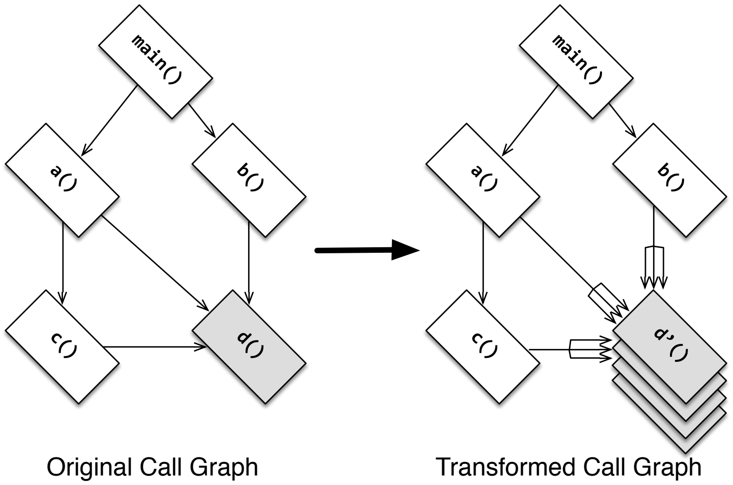


Figure 4: Function call graph before and after replicating the function d().

layout and internals, automatically modifying these aspects of the program implementation hinders development of reliable exploits using techniques such as return-oriented programming. However, software diversity affects not only program layout but also alters program side-effects, such as run time, power usage and cache usage. Even simply re-ordering functions can have a large effect on cache usage and performance since code will be aligned differently in the instruction cache.

Since software diversity affects performance and cache usage, by extension we observed that it could be useful to disrupt or add noise to side channels. However static compile-time or load-time diversity is insufficient, since side-channel attacks are online dynamic attacks and attackers can simply profile the static target binary to learn its runtime characteristics. Re-diversifying and switching to a new variant during execution is also insufficient since side-channel attacks are fast enough to complete between reasonably spaced re-diversification cycles. Figure 3 illustrates the effect of diversification techniques on side channels. While the program trace of the original program leaves a specific footprint on the executing hardware, diversified program variants (labeled as static variant 1 and static variant 2) each have a different footprint. This diversity is likely to thwart offline profiling attacks, but online side-channel attacks that deduce information by monitoring the running program are not affected by these diversification techniques.

We extend previous, mostly static software diversification approaches by dynamically randomizing the control flow of the program while it is running. Rather than statically executing a single variant each time a program unit is executed, we create a program consisting of replicated code fragments with randomized control flow to switch between alternative code replicas at runtime.

In Figure 3, we see the effect of dynamic control-flow diversity in the bottom row, labeled dynamic variant 1. For the trace segment the attacker is interested

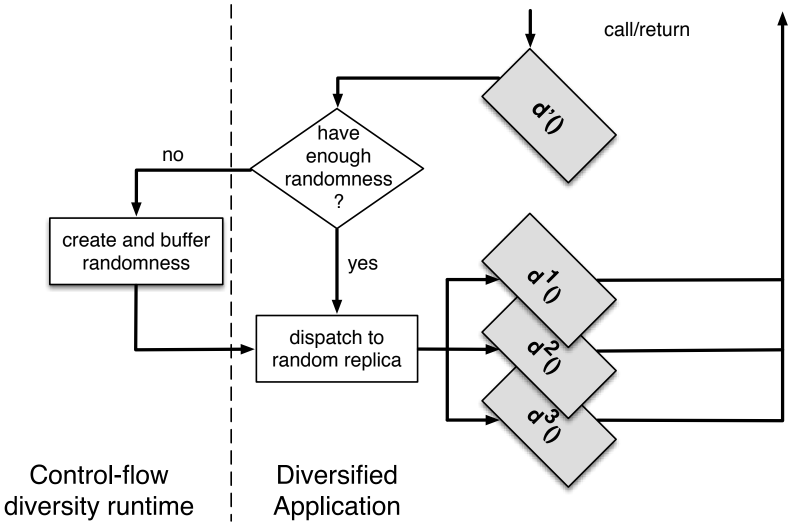

call/return
d'()
have enough randomness ?
no
yes
create and buffer randomness
dispatch to random replica
d1()
d2()
d3()
Control-flow diversity runtime
Diversified Application

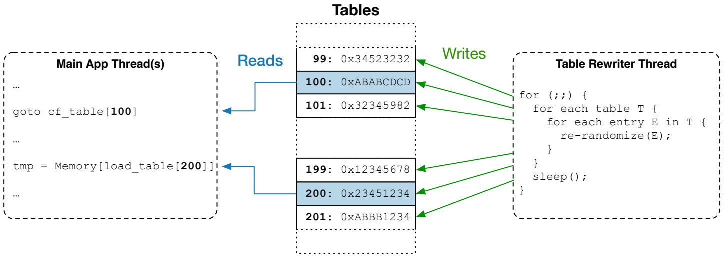
Figure 5: Detailed view of randomized trampoline d'() and interaction with the runtime system.

in, the program can now take numerous different paths, effectively preventing the attacker from constructing a reliable model to infer program execution information from side-channel characteristics, such as timing.

We build our control-flow diversity on a conventional compiler-based diversification system that creates randomized variants of a program fragment, such as a function or a basic block, by applying diversifying transformations. A diversifying transformation preserves program semantics but transforms implementation details. Examples of previously proposed diversifying transformations include insertion of NOP instructions, permutation of function or basic block layout, and randomization of register assignments. In Section 3.1 we discuss a diversifying transformation to illustrate the effects of control-flow diversity on cache side channels, but other transformations could be used to protect against other instances and varieties of side-channel attacks.

To create control-flow diversity, we begin by choosing a set of program fragments (either functions or basic blocks) to transform. If a developer knows that some sections of the program, such as encryption routines, are particularly interesting targets for side-channel attacks, the developer can manually specify this set of program fragments to diversify. In addition, for blanket coverage we can randomly select candidate program fragments. Since randomized control-flow transfers add performance overhead, the software distributor should adjust the percentage of duplicated fragments to balance security and performance.

After choosing a set of functions and/or basic blocks, we clone each chosen program fragment a configurable number of times. We then use different diver-



```
…
goto cf_table[100]
…
tmp = Memory[load_table[200]]
…
```

```
for (;;) {
  for each table T {
    for each entry E in T {
      re-randomize(E);
    }
  }
  sleep();
}
```

Figure 6: Memory layout of runtime address tables, along with pseudocode of the randomization algorithm. The randomization algorithm runs periodically in an infinite loop for the entire duration of the program.

sifying transformations for each clone to create functionally-equivalent replicas that differ in runtime characteristics. The set of transformations applied to each program fragment may include completely different transformations, applications of the same transformation with different parameters, or some combination of both. Figure 4 shows an example of this process applied to a function.

We then integrate these randomized replicas into a program that dynamically chooses control-flow paths at runtime. For each replica, we replace all references to the original fragment with a *randomized trampoline*. As illustrated in Figure 5, whenever the program executes a trampoline it randomly chooses a replica to transfer control to.

We use the SIMD-Oriented Fast Mersenne Twister pseudorandom number generator (PRNG) [37], since the runtime needs to quickly generate random numbers. Although our chosen PRNG is not cryptographically secure, it is sufficient for our purposes, since we assume the attacker cannot extract every control-flow transfer through the noisy side channel. If defending a side channel through which extracting the dynamic control flow and predicting the PRNG stream is easier than extracting the targeted secret information, this PRNG could easily be replaced by a cryptographically secure PRNG. Processor-integrated random number generators would be ideal to fill this role, and, as processors with this capability become widespread, we expect that the processor can fill a randomness buffer instead of using a software PRNG.

3.1 Cache Noise Transformation

In order to produce structurally different but semantically identical variants, we randomly apply diversifying transformations to the program code. These transformations change how a program looks to an observer (who might either read the binary itself, or observe it through side channels) without affecting program semantics. We investigated one specific transformation, inserting cache noise, to disrupt cache side-channel observations. However, this technique is only one example of possible side-channel disrupting transformations. When protecting other side channels, one may need different transformations, e.g., disrupting power observations might require randomly weaving in another unrelated program to ensure that the inserted code is indistinguishable from the original program code.

We initially investigated disrupting the EVICT+TIME attack by randomly inserting NOP instructions into the code. However, after optimizing our randomness generation, we found that NOP instructions do not add enough time fluctuation to disrupt the attack. In addition, NOP instructions have no effect on cache usage, and thus do nothing to affect the PRIME+PROBE attack. We therefore turned our attention to inserting random memory loads, which disrupt both timing and cache snooping side channels.

To ensure that inserted loads have a high likelihood of actually impacting the performance of the targeted program, we want to create loads that evict a specific set of cache lines, specifically those that the target uses. In addition, attempting to read from invalid addresses (such as unallocated regions in the process address space) can potentially crash the target program, stopping the attack. For these reasons, we restrict the loads to a linear region, selected at program load-time. In the case of our AES experiments, this region covers only the AES S-box tables but in general is adjustable for other applications.

Our compiler randomly picks the locations to insert loads at compile time, and the target program itself picks the base and size of the region at load-time during program initialization. We leave the size of each load (in bytes) up to the implementation, and use single-byte loads in our evaluation. While implementing this cache diversification technique, we identified two ways of computing the address accessed by each load instruction: (i) static address and (ii) dynamic address computation.

In the first technique, static address computation, the compiler randomly picks an address (inside the range), then hard-codes it inside the program so the load is the same for every execution:

```
offset = 0x123 // Random constant < region_size
addr = region_base + offset
tmp = Memory[addr] // Volatile load
```

The second technique, dynamic address computation, loads addresses chosen dynamically while the program is running. We extend the same cached random tables used for control-flow diversity described below, and constantly rerandomize this table to contain valid addresses. This results in the following code for inserted load `i`:

```
addr = Memory[random_table[i]]
tmp = Memory[addr] // Volatile load
```

Static address computation requires the region size to be defined at compile time and a global variable `region_base` to be initialized at run time. The background thread for dynamic address computation randomly picks addresses for each table slot using global variables `region_base` and `region_size` that are initialized at run time.

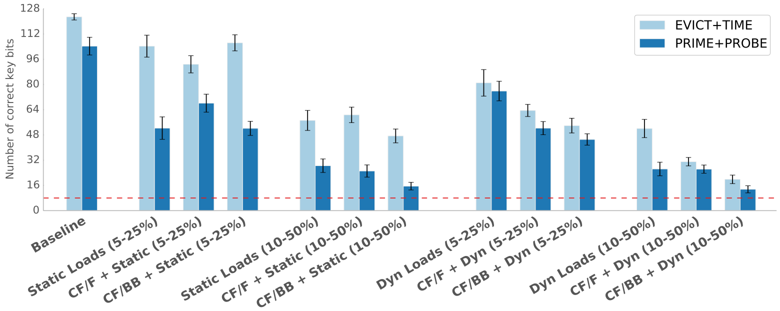


Figure 7: Average accuracy of AES side-channel attacks with our defense. The dashed line shows the expected number of correct bits for randomly chosen keys (8 bits). Error bars represent two standard errors from the mean.

3.2 Table Randomization Optimization

One of our main design goals was to make the randomized trampolines and memory loads be fast enough for practical usage. A naive initial implementation that called a random number generator for every control-flow transfer or memory operation proved to have unacceptably large overhead, even when buffering randomness. We instead chose to store branch targets and memory load addresses in tables and periodically re-randomize this table asynchronously in a background thread. At program startup we create a background thread that repeatedly iterates over all tables and randomizes each entry. Trampolines are then just a single indirect branch through a control-flow cache table, while the memory loads require an extra load from the table. Figure 6 shows the memory layout of the tables and the pseudocode of the table randomization algorithm which runs in the background thread.

Our dynamic control-flow transfer implementation could be further optimized to use inline caching and rewrite static branch instructions rather than an external table in data memory. However, branch targets are rerandomized frequently, so changing code page permissions from executable to writable and back may trump the performance improvement from inline caching. Alternatively, code pages could be left writable and executable, although this increases the risk of a code injection attack and may still be slow if instruction cache flushes are required.

4 Evaluation

To analyze the security and performance characteristics of our techniques in a real-world setting, we evaluated dynamic control-flow diversity as a defense for the two side-channel attacks proposed by Tromer et al. [39] and discussed in Section 2.2. We implemented these attacks targeting the AES-128 encryption routine in libgcrypt 1.6.1, which is the current version of the cryptographic library underlying GnuPG. Since our implementation does not currently support diversification of inline assembly, we disabled the assembly implementation of AES to force libgcrypt to use its standard C implementation. It is worth noting that this is an implementation limitation, and an industrial-strength implementation of our transformations could easily support rewriting of inline assembly as well.

To simplify our attack implementation, we made a slight change to the libgcrypt source code. We added an annotation to each of the targeted tables to force the compiler to align table entries such that no entries crossed a 64 byte cache line boundary. While both attacks could work around this alignment issue with further engineering effort, this change allowed us to more accurately measure the results of our protections.

We performed all security evaluations on an Intel Core 2 Quad Q9300 running Ubuntu 12.04 with Linux kernel 3.5.0. We targeted our attacks at the L2 cache of the processor; the Q9300 contains a 6MB L2 cache split into two halves, with each 3MB half being shared by two of the cores. The cache is 12-way set associative with 64-byte lines and 4096 sets. To minimize system interference, we stopped all unnecessary system daemons and pinned the attack to two cores, where the second core accommodated the background rewriting thread. In addition, to create the most advantageous situation possible for an attacker, our example attacks call the libgcrypt encryption function as a black box in the same process, rather than spawning or communicating with a separate process. Attacks in a more realistic setting would require even more observations to reliably extract the key, and our transformations would create additional uncertainty when coupled with the extra intra-process system noise.

Modern processors implement a cache prefetching algorithm that assumes spatial locality of cache accesses and speculatively loads additional cache lines that the prefetching unit expects might be accessed soon. Prefetching improves performance, especially for algorithms that access long linear regions of memory. However, prefetching negatively impacts cache-based side-channel attacks by introducing the difficulty of determining which lines were loaded by the encryption algorithm and which by the prefetcher. For this reason, we disabled the prefetcher completely on our test machine by setting several configuration bits in machine status registers. While this slightly reduces the overall system performance, it makes attacks much more consistent.

We implemented all transformations and insertion of dynamic control-flow diversity as passes in version 3.3 of the Clang/LLVM compiler framework [29]. These new passes operate at the LLVM intermediate representation (IR) level, and are thus platform-independent.

4.1 Security Evaluation

After testing our example attacks, we empirically found that 5 million iterations of the EVICT+TIME attack and 75 thousand iterations of the PRIME+PROBE attack were sufficient to derive 96% and 82% of the random key bits on average, respectively. Although our attack implementation does not derive the full key in all cases due to random system noise and complex processor variations, this is an implementation concern, and an attacker would likely tune these attacks for increased accuracy.

To ensure that our baseline was accurate, we averaged 50 runs without any diversification, using a new random secret key for each iteration. Since our transformations rely on random choices during compilation, we tested each instance with ten different random seeds and each seed with five random keys (resulting in 50 runs total for each configuration) and report the average accuracy over all seeds and keys for each configuration.

To ensure that functions or basic blocks relevant to the AES encryption implementation were replicated, we manually inspected the libgcrypt implementation and selected nine functions that the program executes for every AES encryption. To collect comparable data for each experiment, we configured our compiler to select all nine functions (or all basic blocks in the selected functions) for replication.

We fixed the number of generated replicas for each program fragment to ten in all cases. We found that further increasing this parameter had little effect on the attack success with the number of iterations we tested. However, increasing the replica count also had no measurable effect on runtime performance, and only a moderate effect on file size. Therefore adding additional variants may be a viable option to combat increasing attacker capabilities.

Security results for both the EVICT+TIME and PRIME+PROBE attacks are found in Figure 7. We label the all static cache load variants with Static and the dynamic variants with Dyn. Control-flow diversity with function and basic-block replicas is labeled respectively with CF/F and CF/BB. We report key recovery in number of bits for clarity, however, it is important to note that both attacks derive the key in nibble-sized increments.

Static Loads Static cache noise at a 5–25% insertion rate had little effect on the EVICT+TIME attack, resulting in 104–108 of 128 key bits recovered. Adding dynamic control-flow diversity to static noise also had little effect, since there is little timing variance between replicas when using static loads. Increasing this percentage to 10–50% had a more pronounced effect. More loads naturally imply that execution will be slower and thus more sensitive to cache collisions.

Dynamic control-flow diversity did have a significant effect on the PRIME+PROBE attack when combined with static cache loads. Function-level dynamic control-flow diversity reduced the correctly key recovered key bits from 52 with static loads to 41, and basic-block level replication furthered reduced this to 31 bits. With 10–50% cache noise insertion, we saw further reduction to 16 key bits correctly recovered using basic-block dynamic control-flow diversity.

Dynamic Loads Dynamic loads had a larger effect on the EVICT+TIME attack. Dynamic cache noise alone at a 5–25% rate reduced the average correctly recovered key bits to 81. Adding dynamic control-flow diversity on top of this further reduced the recovered key bits to 64 and 54 for function-level and basic block-level diversity respectively. At the 10–50% insertion rate we observed similar trends, with CF/BB and dynamic loads reducing the EVICT+TIME key recovery to 20 bits. Dynamic cache loads naturally have a higher performance variation, since they require an extra indirect load to implement runtime dynamic randomness. This results in a more pronounced impact on the EVICT+TIME attack.

We observed similar trends for the PRIME+PROBE attack. While dynamic loads have some effect on the attack by themselves, they are most effective when combined with function or basic-block dynamic control-flow diversity. In the best case (CF/BB + Dyn) we observed an average correct key recovery of only 14 bits. This result is near the theoretical limit of 8 bits where an attacker gains no information from the side channel. Recovering 8 bits of the key is equivalent to an adversary randomly guessing the key by nibbles without side-channel information, since such an adversary has a 1 in 16 chance to guess each nibble correctly and each key nibble is independent for uniform random keys. This expected number of correctly guessed key bits with no knowledge is a lower bound on the accuracy of any side-channel attack, and we show this bound as a dashed line in Figure 7.

Increasing samples To investigate whether the attacks could feasibly overcome our defense by gathering more side-channel observations, we increased the iteration count for both attacks. We found that while the attack accuracy increased marginally with 4x and 8x the number of original attack measurements, a realistic attack is still infeasible. With the CF/BB + Static (10–50%) setting, 4x iterations resulted in average correctness of 70 bits for the EVICT+TIME attack and 34 bits for the PRIME+PROBE attack. 8x iterations resulted in 42 correct key bits on average for the PRIME+PROBE attack. These results indicate that dynamic control-flow diversity is still effective in the presence of better resourced attackers, although it may require a different diversifying transformation to be more effective against the EVICT+TIME attack.

Collecting eight times more samples than in our baseline attack required about five minutes of attack time, resulted in a 1.5GiB data file, and analysis took about an hour on a high end, quad-core c3.xlarge Amazon EC2 instance. In a more realistic situation collecting many more samples than this is likely prohibitive. It is important to remember that our attack is simply encrypting a single block, with no inter-process communication or application overhead. Our tests represent a best-case scenario for an attacker. A realistic attack would target a service which is doing more work than our test attacks, and thus data collection would be far slower and noisier in practice.

4.2 Performance Evaluation

Most existing defenses against cache side-channel attacks, e.g., reloading sensitive tables into cache after every context switch or rewriting encryption algorithms to not use cached tables at all, introduce moderate overheads. Our transformations also marginally increase the cost of AES encryption. However we believe this overhead to be quite reasonable for an automated and general side-channel

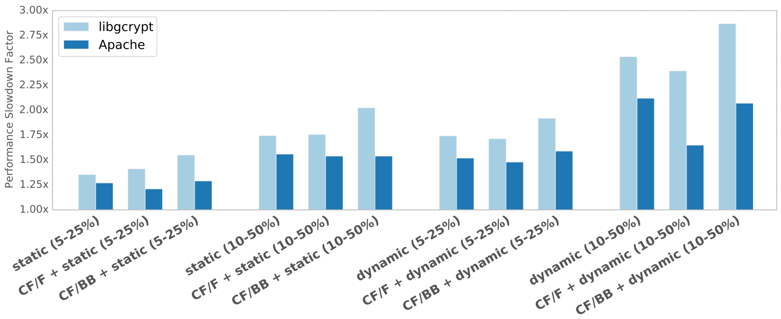


Figure 8: Performance slowdown factor. libgcrypt: AES micro-benchmark encryption performance slowdown, relative to a non-diversified baseline. Apache: slowdown when serving a 4MB file over HTTPS with AES block cipher, relative to a non-diversified baseline.

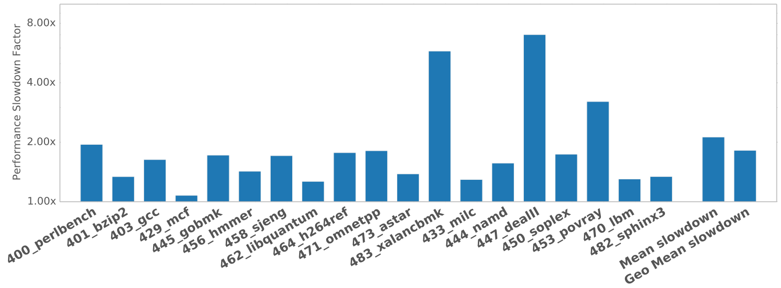


Figure 9: Performance slowdown factor for SPEC CPU2006 with function-level dynamic control-flow diversity on 25% of functions and 10–50% static cache noise inserted in all functions. Y-axis is on a log scale.

defense. To properly quantify this impact, we studied an AES micro-benchmark, a full-fledged service — Apache serving files over HTTPS using AES — and the SPEC CPU2006 benchmark suite.

From this performance analysis, in conjunction with attack success, we found that the optimal trade-off between security and performance is the CF/F + Static Loads setting. The CF/BB + Static Loads setting was slightly more effective, with only a small marginal decrease in performance, and is thus also an ideal candidate setting. Using dynamic loads, while slightly more effective, has a significantly larger performance impact for comparably little marginal security benefit.

AES Micro-benchmark We first measured the increase in time introduced by each transformation with an AES micro-benchmark. We generated ten random different versions of libgcrypt for each set of parameters, ran each version of the AES encryption function five million times on random plaintexts for each of ten

Transformation	File Size (KiB)	Increase Factor
Baseline	657	1.00
Static Loads (5-25%)	657	1.00
CF/F + Static (5-25%)	702	1.07
CF/BB + Static (5-25%)	716	1.09
Dyn Loads (5-25%)	658	1.00
CF/F + Dyn Loads (5-25%)	755	1.15
CF/BB + Dyn Loads (5-25%)	727	1.11
Static Loads (10-50%)	657	1.00
CF/F + Static (10-50%)	766	1.17
CF/F + Static (10-50%)	941	1.43
CF/BB + Static (10-50%)	737	1.12
CF/BB + Static (25@10-50%)	837	1.27
Dyn Loads (10-50%)	660	1.00
CF/BB + Dyn Loads (10-50%)	759	1.15
CF/F + Dyn Loads (10-50%)	784	1.19

Table 1: File size increase for libgcrypt, relative to a non-diversified baseline.

different random keys and measured the number of cycles for each encryption. The first column of each group in Figure 8 shows the slowdown for the libgcrypt micro-benchmark.

We found that using function or basic-block level dynamic control-flow diversity along with static cache noise results in a performance slowdown of 1.76x–2.02x compared to the baseline AES encryption when using 10–50% cache noise insertion. Dynamic cache noise at a 5–25% rate results in similar performance, but 10–50% insertion of dynamic loads has significantly more impact on performance (2.39–2.87x slowdown).

In addition to measuring encryption time, we investigated the impact of our transformations on the size of the encryption library. While desktop disk space is currently plentiful, this is not the case for embedded or mobile systems. Many programs are also distributed over the Internet through communication links that have either bandwidth or data limits. Table 1 shows the impact of our transformations on the size of the libgcrypt shared object.

Application Benchmark In the previous section we measured the performance impact on AES encryption alone, encrypting a single block. However, to get a more realistic picture of the performance impact of our techniques, we also evaluated the performance overhead of dynamic control-flow diversity and our transformations on Apache 2.4.10 serving AES encrypted data. We used the standard apachebench (ab) tool to evaluate performance, connecting over https to an Apache instance using a diversified version of the OpenSSL 1.0.1 library[3].

As seen in the second column of each group in Figure 8, the overall slowdown of our techniques varies from 1.25x for static cache noise to 2.1x for dynamic. The static noise CF/F and CF/BB settings in fact have identical overheads in this test, and we therefore recommend the CF/BB setting for practical applications which consist of more than just block cipher encryption. The overall performance impact is naturally lower than the simple micro-benchmark, since Apache does other processing in addition to encryption. However, this workload is more representative of a real-world application of cryptography and AES in particular.

SPEC CPU2006 To illustrate the effects of our techniques on CPU intensive workloads, we tested with the C and C++ portions of the SPEC CPU2006 benchmark suite. We selected one parameter setting: function-level dynamic control-flow diversity with static noise. However, since SPEC does not have any particular targets for cache side-channel attacks, we applied dynamic control-flow diversity universally over all functions with a 25% probability. We also applied static cache noise over all functions with a probability to insert noise for each instruction chosen randomly for each basic block from the range 10–50%. These parameters represent a worst-case for the CF/F + Dyn setting. To account for random choices, we built and ran SPEC with four different random seeds

As we show in Figure 9, our transformations introduce a 1.82x geometric mean overhead across all benchmarks. The xalancbmk and dealII benchmarks stand out in this test. These particular benchmarks are large, complex C++ programs with many function calls. Since we applied function dynamic control-flow diversity across the entire program in this case, we naturally incur a higher overhead when the program calls many small functions. In practice users of dynamic control-flow diversity should target transformations in only the sections of code which might be vulnerable to a side-channel attack, instead.

5 Discussion

Parameter Settings

In our experiments we determined that a 5–25% insertion percentage range for cache noise instructions is too narrow. Dynamic control-flow diversity works best when replicas have very different runtime behavior, since it relies on switching between replicas with varying side-channel effects. In addition, libgcrypt is mostly straight line code and thus has a relatively low number of functions and basic blocks used for AES encryption. We expect that more complex cryptographic algorithms such as RSA will have more control flow, and thus more opportunity to insert dynamic control-flow diversity and switch between variants.

[3] While we have not tested the effectiveness of the side-channel attack on this library, we believe it would take minimal effort to port the attack to OpenSSL or other table-based AES implementations.

Cache noise, especially the dynamic variant, has an impact on execution time and thus the EVICT+TIME attack. However, this transformation is designed specifically to disrupt the PRIME+PROBE attack by polluting the cache and masking real AES table cache accesses. A transformation targeted at varying the running time of each replica would be more suited to disrupting this attack. We could adapt proposed hardware junk code insertion techniques [25,5] to work with dynamic control-flow diversity by inserting differing code with varying runtimes into each replica.

In the best case, CF/BB + Dyn (10–50%), our EVICT+TIME attack can derive only 4.96 key nibbles, or about 20 key bits. Even with a more performance conscious alternative, CF/BB + Static (10–50%), we still prevent the attacker from finding 80 of 128 key bits. In the PRIME+PROBE attack our experiments show an average of 3.32 correctly recovered key nibbles, or 13.28 key bits, for the CF/BB + Static (10–50%) setting. The remaining approximately unknown key bits are too much to brute-force search, since this would require checking 2^n key guesses, where n is the number of unknown key bits. With this low correctness an attacker is unlikely to even be able to determine which key nibbles are correct, and thus would gain no useful information from the attack. Thus, we conclude that our techniques effectively mitigate the PRIME+PROBE attack, given a realistic attack scenario.

We chose example parameters of ten replicas for each program unit along with 5–25 and 10–50 percent probability of inserting cache noise operations at each instruction as a starting point after initial experimentation. These parameters are representative of a narrow and wider range of insertion. However, these parameters may not represent an ideal trade-off between security and performance. In fact, these parameter settings are not mutually exclusive, e.g., some functions may be diversified with static noise while others get dynamic noise. Some combination of function and basic block replicas may also be useful for some applications. For future work, we propose to develop heuristics for automatic parameter selection through application and attack profiling.

Disabled Cache

Disabling caching of critical memory is an often suggested naive approach to preventing cache side-channel attacks [39]. This approach is attractive since existing commodity processors support selectively disabling page caching, but unfortunately it is prohibitively slow. To verify that this mitigation is impractical, we carefully measured the performance of the AES routine in libgcrypt with caching disabled for the AES lookup tables. This required writing a custom Linux kernel module to map and mark a page of memory as uncacheable using the Page Attribute Table (PAT) available on x86 CPUs. The user mode application, in this case libgcrypt, can then map this page into its address space and store the lookup table into it. This interface, while technically possible, is complex and not available in the standard Linux kernel.

We modified libgcrypt to utilize this approach and tested the same AES micro-benchmark described above. We found that disabling caching on only the single AES lookup page caused the encryption routine to be 75 times slower than normal. Therefore disabling caching, even for a single page, is impractical on modern hardware. We discuss other hardware based cache protections in Section 6, however, these approaches are not available in commodity processors.

Implementation Limitations

For our initial investigation of applying control-flow diversity to side channels, we manually inspected the libgcrypt AES implementation to select nine functions relevant to the encryption algorithm. This simple step required no modification to the original sources, and could be easily automated by supplying only an encryption entry point. We forced our system to replicate these functions and their basic blocks to demonstrate the effectiveness of our techniques in a controlled environment, without the additional complication of having the system automatically select program units for diversification at random. However, this small manual effort was done to arrive at a controlled experiment and is not required to use control-flow diversity. By randomly selecting program units for replication with some configurable probability, our system can probabilistically protect an entire application from side-channel attacks with no manual effort.

Instead of random or manual program unit selection, we believe that side-channel analysis tools such as CacheAudit [15] can guide the selection of the critical program fragments and parameters for diversification. This should eliminate all manual effort while preserving a high level of security.

Related Attacks

Diversifying transformations, such as inserting cache noise instructions, can also be used to perform fine grained code layout randomization. This provides probabilistic protection against return-oriented programming and its variants which makes it realistic to expect that our defense technique can simultaneously defend against two or more fundamentally different classes of attacks. We will pursue this research direction in follow up work as well.

6 Related Work

This paper unites two previously unrelated strands of research: side channels and artificial software diversity. We discuss the related work in each of these areas separately.

6.1 Side Channels

After Kocher described an initial timing side-channel attack on public-key cryptosystems [27], researchers have proposed a multitude of side-channel attacks against cryptographic algorithms. While researchers have proposed many different side-channel vectors ranging from power analysis [26] to acoustic analysis [17], we focus on applying our techniques against timing and cache-based attacks not requiring physical access. Cache-based attacks were first theoretically described by Page [32] in 2002. In 2003, Tsunoo et al. [40] demonstrated cache-based attacks against DES in practice. Bernstein [7] then presented a simple timing attack on AES, along with potential causes of this timing variability, including variable cache behavior and latency. Shortly after, Osvik, Shamir, and Tromer [31,39] presented their attacks on AES, including the two example attacks used in this paper. In addition to the two synchronous attacks we evaluated our techniques against, Osvik et al. also described an asynchronous attack relying only on passively observing encryptions of plaintexts from a known but non-uniform distribution.

Recently, Hund et al. [24] used a cache-based timing side-channel attack to derandomize kernel space ASLR in order to accurately perform code-reuse attacks in the kernel address space. Since we build our system on techniques proven to be effective against code-reuse attacks, our dynamic control-flow diversity with NOP insertion is a perfect fit to defend in depth against this attack.

As side-channel attacks have matured, researchers have proposed numerous defenses using both hardware and software. We will now briefly describe a few of the relevant defenses.

Hardware Defenses Several different methods of preventing side channels at the hardware level have been proposed, with varying degrees of practicality. In the context of differential power analysis attacks, Irwin et al [25] proposed a new stage in the processor execution pipeline which randomly mutates the instruction stream with the assistance of a compiler-generated register liveness map. Among other peephole transformations, this mutation unit adds instructions that do not affect the correct functioning of the program, which are a super-set of our compiler-based NOP insertion transformation. Since our transformations in software are similar to the techniques Irwin et al. applied to differential power analysis, we expect that our technique will apply directly to power analysis attacks as well. Finally, Irwin et al. proposed a new probabilistic branch instruction, *maybe*, that would allow us to efficiently randomize control flow without the use of a random buffer. Ambrose et al. [5] also proposed inserting random instructions but with the added requirement that inserted instructions modify processor state, e.g., registers, so the new instructions are indistinguishable from legitimate program code.

To specifically target cache-based attacks, Page [33] proposed partitioning the cache into disjoint configurable sets so that a sensitive program cannot share cache resources with an attacker. However this would require a radical change to current cache designs. Bernstein [7] suggested the addition of a new CPU instruction to load an entire table into L1 cache and perform a lookup. This approach provides consistent cache access behavior regardless of input, and as such would eliminate cache side channels through table lookups. Wang and Lee [41] also proposed two new hardware cache designs to mitigate cache side channels: PLcache and RPcache. PLcache has the new capability of locking a sensitive cache partition into cache, while RPcache randomizes the mapping from memory locations to cache sets. While these techniques are powerful mitigations against cache side-channel attacks, they all require additional hardware features which major processor vendors are unlikely to implement. In contrast, our techniques require no special hardware support and can be used immediately.

Intel has recently implemented a new hardware instruction to perform encryption and decryption for AES [19]. Since this instruction is data independent, using it instead of a software routine should protect against side-channel attacks on AES. However, this hardware only implements AES, and thus we still need defensive measures to protect other cryptographic algorithms.

Software Defenses The ideal defense against side-channel attacks is to modify the sensitive program so that it has no input-dependent side-effects, however this is an extremely labor-intensive solution and is often infeasible. Developers generally take this approach to removing timing side channels by creating algorithms that run in constant-time regardless of inputs. Bernstein [7] strongly recommends this approach, while cautioning that software which the programmer expected to run in constant time may not do so due to hardware complexity.

Page [34] suggested manually adding noise to encryption to make cache side-channel attacks more difficult in a manner conceptually similar to our automatic randomizing transformations. For instance, Page manually inserted garbage instructions and random loads into the encryption routine to combat timing and trace based attacks respectively. Page's work is a form of obfuscation rather than diversification since all users run the same binaries with the same runtime control-flow. Our combination of control flow randomization and garbage code insertion *simultaneously* defends against code reuse attacks and side channels whereas garbage code in itself does not protect against side channels and Page's transformations do not protect against code reuse.

Brickell et al. [8] proposed the use of compressed and randomized tables for AES that would alleviate cache-based attacks. However, this implementation process requires manually rewriting the AES implementation and is specific to the operation of AES.

Cleemput et al. [9] proposed defenses that do not require manual program modification. In particular, they described the use of compiler transformations to reduce timing variability. Our approach, while also compiler-based, seeks to mask variability rather than remove it entirely, since opportunities to automatically eliminate variable-time routines are limited.

In their recent paper addressing side-channel attacks in the context of virtualized cloud computing, Zhang and Reiter [45] proposed periodically scrubbing shared caches used by sensitive processes. This scheme potentially breaks cache snooping by a time-shared process on the same core, but will not necessarily combat cache attacks in a Simultaneous Multithreading (SMT) context or continuous power analysis attacks. Since our random decision points are more fine grained than the scrubbing interval, our techniques have greater potential against these fine-grained attacks, although this would require more investigation. In addition, control-flow diversity does not depend on any resources outside the program and is thus applicable in situations without hypervisors, such as embedded software.

Finally, Tromer et al. [39] mention adding noise to memory accesses with spurious accesses to decrease the signal available to the attacker. Effectively, our technique accomplishes this goal in a general way that could be extended to other side channels, and we provide a concrete evaluation showing its effectiveness in practice. Since adding replicas exponentially increases the number of possible execution traces, we can ratchet our defense up sufficiently so that an attacker cannot feasibly collect and analyze enough samples.

6.2 Artificial Software Diversity

The literature on artificial software diversity is extensive; we limit ourselves to the work most closely related to ours. Larsen et al. provides a comprehensive systematization of approaches to artificial software diversity [28]. Cohen initially pioneered software diversity as a protection against reverse engineering [10] and was first to suggest garbage code insertion and transformations that obscure the actual control flow. Collberg et al. [11] extended these ideas into a broader set of obfuscating transformations against reverse engineering attacks and introduced the notion of opaque predicates [13]. While opaque predicates usually refer to predicates that have a known outcome at obfuscation time but are expensive to decide afterward via static analysis, Collberg et al. also mention "variable" opaque predicates that flip-flop between true and false at runtime. These ideas were evaluated in depth by Anckaert et al. [6] as a defense against reverse engineering, by Collberg et al. [12] in context of client-side tampering of networked systems, and by Coppens et al. [14] to prevent reverse engineering of patches. Our work differs in its use of control flow randomization: we use it to switch among implementation variants (replicas) with fine-granularity—not as a randomizing transformation in itself. Furthermore, we aim to thwart side-channel attacks rather than reverse engineering.

Several diversified defenses against code reuse attacks have dynamic aspects. Giuffrida et al. [18] presented a compiler-based approach that periodically rerandomizes services in a microkernel OS while it is running. Live rerandomization works by periodically transferring the application state from one process to another such that the old and new processes run diversified variants of the same input program. While this provides excellent protection against code reuse attacks, the rerandomization overhead prevents the fine granularity our approach efficiently supports.

Hiser et al. [21] performed fine-grained code layout randomization using a process virtual machine. The approach uses a code cache that leads to predictable program traces and might constitute a side channel in itself. Homescu et al. [22] diversifies just-in-time compiled code and similarly caches translated code to improve performance. Shioji et al. [38] introduced "code shredding" that embeds random checksums in pointers to thwart control-flow hijacking. To improve performance and add randomness, checksums are not masked out before the pointer values are used in control flow transfers. Rather, the entire code section is replicated in process memory to make the targets of checksummed pointers valid. Our use of code replication is more flexible because our granularity can vary at the function or basic block level and has a lower memory overhead as a result. Our performance overhead is also much lower since our compiler-based approach avoids the overheads associated with binary rewriting; Shioji report overheads ranging from 3x to 26x on Bzip2 1.0.5.

Novark and Berger secure the heap against memory management errors via a randomizing memory allocator [30]. Allocations are placed randomly in memory and stay in place until their deallocation. Freed pages are overwritten with random data. While this can interfere with side-channel attacks, attackers can sample the victim process arbitrarily many times between memory allocator activations.

Summing up, our work is the first to use software diversity to mitigate cache side-channel attacks. Previous diversification approaches comprise one or more randomizing code transformations. Our approach consists of a runtime randomization mechanism to dynamically vary execution characteristics *in addition to* a set of randomizing code transformations.

7 Conclusion and Outlook

We provide the first evaluation of software diversity as a side-channel mitigation. To that end, we developed dynamic control-flow diversity which performs fine-grained program trace randomization. Our technique does not require source code modification or specialized hardware so it can be automatically applied to existing software. We have implemented a prototype diversifier atop LLVM and rigorously evaluated the performance of our techniques using modern, realistic cache side-channel attacks in a setting that favors attackers. Our experimental evaluation shows that our technique mitigates cryptographic side channels with high efficacy and moderate overhead of 1.5–2x in practice, making it viable for deployment.

Beyond the cryptographic side-channel problem addressed in this paper, we expect that control-flow diversity is simultaneously effective against other implementation-dependent attacks, including code reuse and reverse engineering. We plan to explore this in future work.

Acknowledgments

We thank the anonymous reviewers for their insightful comments and suggestions. We are also grateful for helpful feedback from Mathias Payer and Mark Murphy.

This material is based upon work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contracts D11PC20024 and N660001-1-2-4014, and generous gifts from Mozilla and Oracle. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research

Projects Agency (DARPA), its Contracting Agents, or any other agency of the U.S. Government.

References

1. Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS '12). ACM (2012)
2. Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC '12) (2012)
3. Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P '12) (2012)
4. Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS '13). ACM (2013)
5. Ambrose, J.A., Ragel, R.G., Parameswaran, S.: RIJID: random code injection to mask power analysis based side channel attacks. In: Proceedings of the 44th Design Automation Conference (DAC '07). pp. 489–492. ACM (2007)
6. Anckaert, B., Jakubowski, M., Venkatesan, R., Bosschere, K.D.: Run-time randomization to mitigate tampering. In: Proceedings of the 2nd International Workshop on Security (IWSEC '07). pp. 153–168 (2007)
7. Bernstein, D.J.: Cache-timing attacks on AES. Preprint (2005)
8. Brickell, E., Graunke, G., Neve, M., Seifert, J.P.: Software mitigations to hedge AES against cache-based software side channel vulnerabilities. Cryptology ePrint Archive, Report 2006/052 (2006)
9. Cleemput, J.V., Coppens, B., De Sutter, B.: Compiler mitigations for time attacks on modern x86 processors. ACM Transactions on Architecture and Code Optimization 8(4), 23:1–23:20 (Jan 2012)
10. Cohen, F.: Operating system protection through program evolution. Computers and Security 12(6), 565–584 (Oct 1993)
11. Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscating transformations. Tech. Rep. 148, Department of Computer Science, University of Auckland, New Zealand (1997)
12. Collberg, C.S., Martin, S., Myers, J., Nagra, J.: Distributed application tamper detection via continuous software updates. In: Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC '12) [2], pp. 319–328
13. Collberg, C.S., Thomborson, C.D., Low, D.: Manufacturing cheap, resilient, and stealthy opaque constructs. In: Proceedings of the 25th ACM Symposium on Principles of Programming Languages (POPL '98). pp. 184–196 (1998)
14. Coppens, B., De Sutter, B., Maebe, J.: Feedback-driven binary code diversification. ACM Transactions on Architecture and Code Optimization 9(4), 24:1–24:26 (Jan 2013)
15. Doychev, G., Feld, D., Köpf, B., Mauborgne, L., Reineke, J.: Cacheaudit: A tool for the static analysis of cache side channels. In: Proceedings of the 22nd USENIX Security Symposium. pp. 431–446 (2013)
16. Forrest, S., Somayaji, A., Ackley, D.: Building diverse computer systems. In: Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HOTOS '97). pp. 67–72 (1997)
17. Genkin, D., Shamir, A., Tromer, E.: RSA key extraction via low-bandwidth acoustic cryptanalysis. In: Garay, J.A., Gennaro, R. (eds.) Advances in Cryptology (CRYPTO '14). Lecture Notes in Computer Science, Springer Berlin Heidelberg (Jan 2014)

18. Giuffrida, C., Kuijsten, A., Tanenbaum, A.S.: Enhanced operating system security through efficient and fine-grained address space randomization. In: Proceedings of the 21st USENIX Security Symposium. pp. 475–490 (2012)
19. Gueron, S.: Intel advanced encryption standard (AES) instructions set. Intel White Paper (2010)
20. Gullasch, D., Bangerter, E., Krenn, S.: Cache games - bringing access-based cache attacks on AES to practice. In: Proceedings of the 32nd IEEE Symposium on Security and Privacy (S&P '11). pp. 490–505 (2011)
21. Hiser, J., Nguyen-Tuong, A., Co, M., Hall, M., Davidson, J.W.: ILR: Where'd my gadgets go? In: Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P '12) [3], pp. 571–585
22. Homescu, A., Brunthaler, S., Larsen, P., Franz, M.: librando: Transparent code randomization for just-in-time compilers. In: Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS '13) [4], pp. 993–1004
23. Homescu, A., Neisius, S., Larsen, P., Brunthaler, S., Franz, M.: Profile-guided automatic software diversity. In: Proceedings of the 11th IEEE/ACM International Symposium on Code Generation and Optimization (CGO '13). pp. 1–11 (2013)
24. Hund, R., Willems, C., Holz, T.: Practical timing side channel attacks against kernel space ASLR. In: Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P '13) (2013)
25. Irwin, J., Page, D., Smart, N.: Instruction stream mutation for non-deterministic processors. In: Proceedings of the 13th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP '02). pp. 286–295 (2002)
26. Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) Advances in Cryptology (CRYPTO '99), pp. 388–397. No. 1666 in Lecture Notes in Computer Science, Springer Berlin Heidelberg (Jan 1999)
27. Kocher, P.C.: Timing attacks on implementations of diffie-hellman, RSA, DSS, and other systems. In: Koblitz, N. (ed.) Advances in Cryptology (CRYPTO '96), pp. 104–113. No. 1109 in Lecture Notes in Computer Science, Springer Berlin Heidelberg (Jan 1996)
28. Larsen, P., Homescu, A., Brunthaler, S., Franz, M.: SoK: automated software diversity. In: Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P '14). pp. 276–291 (2014)
29. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the 2nd IEEE/ACM International Symposium on Code Generation and Optimization (CGO '04). pp. 75–87 (2004)
30. Novark, G., Berger, E.D.: Dieharder: securing the heap. In: Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS '10). pp. 573–584. ACM (2010)
31. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: The case of AES. In: Pointcheval, D. (ed.) Topics in Cryptology (CT-RSA '06), pp. 1–20. No. 3860 in Lecture Notes in Computer Science, Springer Berlin Heidelberg (Jan 2006)
32. Page, D.: Theoretical use of cache memory as a cryptanalytic side-channel. Cryptology ePrint Archive, Report 2002/169 (2002)
33. Page, D.: Partitioned cache architecture as a side-channel defence mechanism. Cryptology ePrint Archive, Report 2005/280 (2005)
34. Page, D.: Defending against cache-based side-channel attacks. Information Security Technical Report 8(1), 30–44 (2003)
35. Pappas, V., Polychronakis, M., Keromytis, A.D.: Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In: Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P '12) [3], pp. 601–615

36. Ristenpart, T., Tromer, E., Shacham, H., Savage, S.: Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In: Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS '09). pp. 199–212. ACM (2009)
37. Saito, M., Matsumoto, M.: SIMD-Oriented fast mersenne twister: a 128-bit pseudorandom number generator. In: Keller, A., Heinrich, S., Niederreiter, H. (eds.) Monte Carlo and Quasi-Monte Carlo Methods 2006, pp. 607–622. Springer Berlin Heidelberg (Jan 2008)
38. Shioji, E., Kawakoya, Y., Iwamura, M., Hariu, T.: Code shredding: byte-granular randomization of program layout for detecting code-reuse attacks. In: Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC '12) [2], pp. 309–318
39. Tromer, E., Osvik, D.A., Shamir, A.: Efficient cache attacks on AES, and countermeasures. Journal of Cryptology 23(1), 37–71 (Jan 2010)
40. Tsunoo, Y., Saito, T., Suzaki, T., Shigeri, M., Miyauchi, H.: Cryptanalysis of DES implemented on computers with cache. In: Walter, C.D., Ç. K. Koç, Paar, C. (eds.) Cryptographic Hardware and Embedded Systems (CHES '03), pp. 62–76. No. 2779 in Lecture Notes in Computer Science, Springer Berlin Heidelberg (Jan 2003)
41. Wang, Z., Lee, R.B.: New cache designs for thwarting software cache-based side channel attacks. In: Proceedings of the 34th International Symposium on Computer Architecture (ISCA '07). pp. 494–505. ACM (2007)
42. Wartell, R., Mohan, V., Hamlen, K.W., Lin, Z.: Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In: Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS '12) [1], pp. 157–168
43. Yarom, Y., Falkner, K.: Flush+reload: a high resolution, low noise, L3 cache side-channel attack. Cryptology ePrint Archive, Report 2013/448 (2013)
44. Zhang, Y., Juels, A., Reiter, M.K., Ristenpart, T.: Cross-VM side channels and their use to extract private keys. In: Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS '12) [1], pp. 305–316
45. Zhang, Y., Reiter, M.K.: Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In: Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS '13) [4], pp. 827–838

What every compiler writer should know about programmers or “Optimization” based on undefined behaviour hurts performance

M. Anton Ertl[⋆]

TU Wien

Abstract. In recent years C compiler writers have taken the attitude that tested and working production (i.e., *conforming* according to the C standard) C programs are buggy if they contain undefined behavior, and they feel free to compile these programs (except benchmarks) in a way that they no longer work. The justification for this attitude is that it allows C compilers to optimize better. But while these “optimizations” provide a factor 1.017 speedup with Clang-3.1 on SPECint 2006, for non-benchmarks it also has a cost: if we go by the wishes of the compiler maintainers, we have to “fix” our working, conforming C programs; this requires a substantial effort, and can result in bigger and slower code. The effort could otherwise be invested in source-level optimizations by programmers, which can have a much bigger effect (e.g., a factor > 2.7 for Jon Bentley’s traveling salesman program). Therefore, optimizations that assume that undefined behavior does not exist are a bad idea not just for security, but also for the performance of non-benchmark programs.

1 Introduction

Compiler writers are sometimes surprisingly clueless about programming. For example, the first specification for Fortran states: “no special provisions have been included in the FORTRAN system for locating errors in formulas” and “FORTRAN should virtually eliminate coding and debugging”, and this approach to error checking made it into the finished product; there were also no programmer-defined functions in the original design, but that was fixed before release [Bac81].

More recently, people have meant one of several different programming languages when they wrote about C; for clarity, we will use different names for these programming languages:

[⋆] Correspondence Address: Institut für Computersprachen, Technische Universität Wien, Argentinierstraße 8, A-1040 Wien, Austria; anton@mips.complang.tuwien.ac.at

$C^\star$ A language (or family of languages) where language constructs correspond directly to things that the hardware does. E.g., `*` corresponds to what a hardware multiply instruction does. In terms of the C standard, *conforming programs* are written in $C^\star$.

"C" The subset of $C^\star$ that excludes all undefined behavior according to the C standard. In C-standard terms, programs written in "C" are *strictty conforming programs*.

C_{bench} A subset of $C^\star$ and (slight) superset of "C" that includes the benchmarks considered relevant by the compiler maintainers (e.g., the SPEC CPU benchmarks).

We look at the differences between $C^\star$ and "C" in Section 2.

Production programmers typically think in $C^\star$ when programming, and as a result, they program in $C^\star$, so most production code is not written in "C" (see Section 2). The cluelessness of many recent C compiler maintainers is that they officially support only "C" and officially feel free to compile any source code that performs undefined behavior into arbitrary machine code,[1] even programs that were tested and worked as intended with earlier versions of the same compiler. As John Regehr puts it: "A sufficiently advanced compiler is indistinguishable from an adversary."[2] Inofficially, these compilers support C_{bench}, but that is of little benefit to other programs.

Unlike the authors of the first Fortran compiler, the current C compiler maintainers stubbornly insist on their view.[3] The reason for this seems to be that they evaluate their work through benchmark results of a certain set of benchmarks; by only having to compile these benchmark programs as intended, they want to give their optimizer freedom to produce better benchmark results.

Better benchmark numbers alone are a weak justification for not compiling production programs as intended, so the C compiler maintainers also claim that these "optimizations" give speedups for other programs. However, that would require programmers to convert their $C^\star$ programs to "C" first, a process which can produce worse code (Section 3), and more importantly, requires an effort that would be much more effective if directed at source-level optimizations. We look at the performance benefits of source-level optimizations in Section 4 and compare it to the differences seen from "optimizations" based on undefined behavior.

Section 5 discusses what compiler writers, standards committees, programmers, and researchers can and should do about these issues.

2 The difference between $C^\star$ and "C"

Both languages have the same syntax and the same static semantics; the difference is in the run-time semantics.

[1] The classical intimidation was "may format your hard drive", but recently "make demons fly out of your nose" (in short: *nasal demons*) seems to be more popular.

[2] http://blog.regehr.org/archives/970

[3] http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html

C* maps language elements to corresponding hardware features, and is consistent about this, at least on the platform. The actual behavior at run-time may be different between platforms, lead to an exception, or worse, overwriting of an unrelated data structure (and it is thinkable, although very improbable, that this eventually results in a formatted hard disk on some platforms), but can always be explained by a sequence of hardware steps corresponding to the source program.

"C" is a subset of C* that tries to specify what is portable between different hardware platforms (including some that have died out) and between different C compilers. Therefore it specifies that the behavior of many language elements under some circumstances is *undefined*, implementation-defined[4], or similar. The original idea was that C compilers implement C*, and that undefined behavior gives them wiggle room to choose an efficient hardware instruction; e.g., for `<<` use the hardware's shift instruction, and differences between different architectures for some parameters result in not defining the behavior in these cases. But in recent years, compiler maintainers have gone beyond that and "optimized" programs based on the assumption that undefined behavior does not happen, in ways that do not correspond to any direct mapping from language element to the actual hardware; e.g., they "optimize" a bounded loop into into an infinite loop.

There are 203 undefined behaviors listed in appendix J of the C11 standard (up from 191 in C99). And these are not just obscure corner cases that do not occur in real programs, on the contrary, it is likely that most, if not all, production programs exhibit undefined behavior; even in GCC and LLVM itself (i.e., the pinnacles of the church of "C"), undefined behior has been found even when just compiling an empty C or C++ program with optimizations turned off.[5] Standards conformance was a requirement to be considered for inclusion in SPEC CPU 2006, yet 6 out of 9 C programs perform C99-undefined integer operations [DLRA12], and these are not the only undefined behaviors around by far.

2.1 Optimization* and "Optimization"

An optimization is a program transformation that preserves the observable behavior of the program and hopefully results in a program that consumes fewer resources (run-time and code size are typical metrics).

There are a large number of effective optimizations that can be used on C* programs (called optimizations* in the following), e.g., strength reduction, inlining, or register allocation. A simple example would be to optimize a multiplication by 5 into a `lea` instruction on the AMD64 architecture.

GCC and LLVM have been adding "optimizations" based on undefined behavior in "C". These work by assuming that the program exhibits no undefined

[4] And apparently the implementation is allowed to define the implementation-defined behavior as undefined.

[5] http://blog.regehr.org/archives/761

behavior, and deriving various “facts” from this assumption, e.g., about values of variables, propagates these “facts” throughout the program, and uses them in other places for “optimizations”. An example is the following function from the SPEC benchmark 464.h264ref:

```
int d[16];

int SATD (void)
{
  int satd = 0, dd, k;
  for (dd=d[k=0]; k<16; dd=d[++k]) {
    satd += (dd < 0 ? -dd : dd);
  }
  return satd;
}
```

This was “optimized” by a pre-release of gcc-4.8 into the following infinite loop:

```
SATD:
.L2:
jmp .L2
```

What happened? The compiler assumed that no out-of-bounds access to d would happen, and from that derived that k is at most 15 after the access, so the following test `k<16` can be “optimized” to `1` (true), resulting in an endless loop. Then the compiler sees that the return is now unreachable, that satd is dead, that dd is dead, and k is dead, and optimizes the rest away.

The GCC maintainers subsequently disabled this optimization for the case occuring in SPEC,[6] demonstrating that, inofficially, GCC supports $\mathrm{C}_{\mathrm{bench}}$, not “C”.

This kind of “optimization” certainly changes the observable behavior, but its advocates defend it by saying that programs broken by these “optimizations” have been buggy all along. Given the widespread occurence of undefined behavior in production code, this means that the compiler maintainers feel free to compile pretty much every production program in a way that it behaves differently than intended and different from the code produced by an earlier version of the same compiler and tested successfully.

If undefined behaviour is so widespread, why do we notice code broken by “optimizations” only occasionally? Undefined behavior is a run-time property, and only a small portion of these occurs in a way that can be turned into (compile-time) “optimization”. Many of of these cases are checks for special cases that do not occur in most executions, such as a check for a buffer overflow; so “optimizing” such checks away may go unoticed unless the program tests for

[6] It still strikes for other programs, see gcc bug 66875.

these specific checks; and such testing may be hard to achieve in larger programs, because 100% test coverage is hard to achieve.

Wang et al. [WZKSL13] have written a static program analyzer that tries to find code that may be “optimized” away in “C”, but not optimized* away in C*. They found that 3,471 packages out of 8,575 packages in Debian Wheezy contain a total of about 70,000 such pieces of code (as far as their checker could determine). In most cases “optimizing” these pieces of code away would result in code different from what the programmer intended (programmers rarely write code that they intend to be optimized away). These numbers are pretty alarming, but probably far lower than the number of undefined behaviors and packages containing them.

More on optimization* Actually I was a little bit too cavalier about optimizations* not changing the observable behaviour of C* programs. It is actually possible to write programs in C* where any change in the generated code produces an observable difference, e.g., a program that outputs the bytes in its object code.

But C*programmers would not complain about that. They generally don’t expect this level of stability. After all, the bytes in the object code change every time there is a change in the program, e.g., due to a bug fix or a new feature.

What they do expect is, in the first order, the direct results of language elements must not change if they are observable (i.e., influence output or exceptions/signals). So optimization such as strength reduction, dead code elimination, or jump optimization are fine.

Register allocation can change the results of accesses to uninitialized variables. This is also accepted by C* programmers: they don’t rely on the values of uninitialized variables, because these values often change during maintenance even in the absence of register allocation.[7]

Of course, “optimization” defenders like to tell stories about bug reports about changes in behavior due to, e.g., changed values of uninitialized variables when optimization is turned on, implying that there is no difference between optimization* and “optimization”. But my guess is that in most such cases the bug reporters were not aware that the reason for the breakage is an uninitialized variable, and once they are aware of that, they are likely to change the program to avoid using such values, because the program would also be likely to break on maintenance.

And these kinds of reports are not that frequent: Of the 25 bug reports for gcc components rtl-optimization and tree-optimization resolved or closed between 2015-07-01 and 2015-07-16, three were marked as invalid, and all three were due to “optimizations”, none due to optimizations*; Bug 66875 was very similar to the SATD example shown above (but the bug reporter accepted that his program is buggy due to the out-of-bounds array access), and Bugs 66804 and 65709 are both due to gcc using aligned rather than unaligned instructions

[7] Alternatively, a compiler striving for a more deterministic behaviour could initialize all uninitialized variables to a fixed value, at a small cost in performance.

when autovectorizing code with unaligned accesses on the AMD64 architecture (which normally does not impose alignment restrictions).

"Optimization" and benchmarks Compiler maintainers justify "optimizations" with performance, and they have benchmark results to prove it; actually, not really (see below). But if they had such benchmark results, would they really prove anything? There are significant differences between production programs and benchmarks:

- Production programs are maintained, including performance tuning by programmers if desired.
- Benchmark programs are fixed, and are normally not changed anymore. Therefore they cannot benefit from further source-level optimizations by programmers.
- Benchmark programs are fixed, and therefore exempt from the policy of compiler maintainers that it is ok to break code with undefined behavior, as we have seen from the released gcc not breaking `SATD()` from SPECint. Benchmark programs are not rewritten to eliminate undefined behaviors as compiler maintainers demand of non-benchmark programs, and therefore do not suffer from the worse code that such rewrites can cause (see Section 3).

Therefore, even if "optimizations" produce speedups for benchmarks, that does not say anything about the performance effect of enabling "optimizations" on production code.

But do "optimizations" actually produce speedups for benchmarks? Despite frequent claims by compiler maintainers that they do, they rarely present numbers to support these claims. E.g., Chris Lattner (from Clang) wrote a three-part blog posting[8] about undefined behavior, with most of the first part devoted to "optimizations", yet does not provide any speedup numbers. On the GCC side, when asked for numbers, one developer presented numbers he had from some unnamed source from IBM's XLC compiler, not GCC; these numbers show a speedup factor 1.24 for SPECint from assuming that signed overflow does not happen (i.e., corresponding to the difference between `-fwrapv` and the default on GCC and Clang).

Fortunately, Wang et al. [WCC+12] performed their own experiments compiling SPECint 2006 for AMD64 with both gcc-4.7 and clang-3.1 with default "optimizations" and with those "optimizations" disabled that they could identify, and running the results on a on a Core i7-980. They found speed differences on two out of the twelve SPECint benchmarks: 456.hmmer exhibits a speedup by 1.072 (GCC) or 1.09 (Clang) from assuming that signed overflow does not happen. For 462.libquantum there is a speedup by 1.063 (GCC) or 1.118 (Clang) from assuming that pointers to different types don't alias. If the other benchmarks don't show a speed difference, this is an overall SPECint improvement by a factor 1.011 (GCC) or 1.017 (Clang) from "optimizations".

[8] http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html

2.2 The intended meaning of programs

Why do programmers program in $C^\star$, not in “C”? Programmers are usually not language lawyers, nor should they be required to. They usually learn a programming language by reading introductory books, by looking at programs written in the language, and by trying out what the compiler does for various programs, and build a relatively simple model from that, possibly incorporating other knowledge (e.g., about hardware).

GCC maintainers have claimed that they don’t know what the intended meaning of programs with undefined behaviour is. However, their own compiler is evidence against this claim. It compiles exactly the code intended by the programmer unless it sees enough of the program to derive “facts” that are then used for “optimization”. E.g., in the SATD example, if the compiler did not know the number of elements of d (e.g., because d is defined in a different compilation unit), the compiler would not “optimize” the code and would compile it exactly as intended.

And that’s what programmers expect: In the normal case a read from an array (even an out-of-bounds read) performs a load from the address computed for the array element; the programmer expects that load to produce the value at that address, or, in the out-of-bounds case, it may also lead to a segmentation violation (Unix), general protection fault (Windows), or equivalent; but most programmers do not expect it to “optimize” a bounded loop into an endless loop.

The expectations of programmers are reinforced, and partially formed, by the behaviour of compilers. Most of the time programmers experience the compiler without “optimizations” kicking in.

So, contrary to claims by “optimization” advocates[9], the compiler does not need psychic powers to determine the intent of the programmer in case of undefined behaviour, it knows the intent already.

A common assumption is that this expected behaviour corresponds to the behaviour when turning off optimization. That is often, but not always the case. In particular, at least some versions of GCC “optimize” `x+1<=x` even with `-O0`.

The programmers’ model may or may not include some caveats (such as different sizes of pointers between 32-bit and 64-bit platforms, or that one should avoid unaligned accesses on some hardware), but not all the fine details of undefined behavior in the C standard. And a programmer who programs for just one platform will often not feel compelled to heed portability caveats even if he knows them. So, we cannot exclude non-portable code from $C^\star$.

What is more likely to not be in $C^\star$ is non-maintainable assumptions (such as the values of uninitialized variables). Still, be very cautious before introducing new optimizations that rely on the assumption that a certain usage is unmaintainable and therefore won’t occur.

Coming back to the psychic powers question, while the compiler cannot know which undefined behaviours a particular program exhibits, it is safe to use a conservative approximation (i.e. $C^\star$). One can ask whether the optimization under

[9] news:<hO2dnSk5W8n4p57OnZ2dnUVZ_qSdnZ2d@supernews.com>

consideration would break things that are likely to be stable across maintenance, or one can use the large corpus of working free software to get an idea which undefined behaviours do occur in production programs. Deriving such knowledge from bug reports is also an option, if all else fails; e.g., if there are several bug reports on alignment problems from autovectorization, obviously unaligned accesses are used by programmers on architectures that support them, and therefore autovectorization should by default use SIMD instructions without alignment restrictions on these architectures.

2.3 Security

I have read arguments that use security to justify compiling programs with undefined behavior different from what was intended. While this paper focuses on the performance and optimization claims, this section discusses this argument briefly.

The argument tends to go something like this: Because there are out-of-bounds array accesses that are security vulnerabilities (in particular, buffer overflow vulnerabilities), and out-of-bounds array accesses are undefined behaviors, it is good for security to compile undefined behaviors differently from the expectations of the programmer. There may be the assumption in this argument that programmers are encouraged to produce fewer out-of-bounds array accesses if compilers do that.

That argument is wrong for several reasons:

- Not all vulnerabilities perform undefined behavior (e.g., privilege escalation or SQL injection).
- Undefined behavior is just as hard to find as vulnerabilities (both usually only show up at run-time for certain corner-case inputs), so encouraging programmers to eliminate undefined behaviors will not make it easier to find vulnerabilities. For a given amount of effort, it is more likely that programmers will find more vulnerabilities if they focus on vulnerabilities than if they also have to look out for undefined behavior.
- Buffer overflow vulnerabilities will typically not be “optimized” into code different from what the programmer intended, because the compiler usually cannot statically determine for such code that the out-of-bounds access happens. If the compiler can determine it, reporting it to the programmer makes much more sense than to “optimize” the code.
- Not all undefined behaviors result in a vulnerability; on the contrary, some security checks perform undefined behavior, and have been “optimized” away [WCC$^+$12,WZKSL13]. There the “optimizing” compiler created a vulnerability that was not present in the C* source code.

2.4 Expressive power

One might think that one can express in "C" all that one can express in C*, but to my surprise that's not the case. So while we are doing a detour from the performance theme of the rest of the paper, let us look at an example of that.[10]

Given that C was designed as a systems programming language, implementing a simple function like memmove() in terms of lower-level constructs should be possible in C, e.g, as follows:

```
void *memmove(void *dest, const void *src, size_t n) {
  if (dest<src)
    memcpy_pos_stride(dest,src,n);
  else
    memcpy_neg_stride(dest,src,n);
}
```

where memcpy_pos_stride() copies the lower-addressed bytes of src before higher-addressed bytes, and memcpy_neg_stride() copies the higher-addressed bytes before the lower-addressed ones (I believe that these helper functions can be implemented in "C", but I also believed that of memmove() before looking closely into it).

However, this implementation is C*, but not "C", because p<q is not defined in "C" if p and q point to different objects. Apparently the reason is to allow more efficient implementation of these operations on segmented architectures (by comparing only the offsets). However, it is likely that both variants above work in C* on such platforms, too, because the result of the comparison does not matter if the pointers point to different objects/segments (in that case either memcpy variant will be correct).

However, a "C" compiler might assume that src and dest point to the same object, propagate that "knowledge" to all users of memmove(), and perform various "optimizations" based on that, even on hardware with a flat address space.

Another memmove() implementation is based on malloc()ing an intermediate copy, but malloc() can fail, whereas the standard memmove() cannot.

3 Code quality

Your production program is not a benchmark, so compiler maintainers demand that you change your program into a "C" program. Let's see what this can do to the code quality of a simple example.

We want to determine whether a variable x of some signed integer type is the smallest value that type can hold. A succinct way to express this in C* (and in Java) is x-1>=x. This may seem puzzling if you think about these types as

[10] For a longer discussion, see news:<2015May1.155805@mips.complang.tuwien.ac.at> ff.

mathematical integers, but note that these are bounded types; if x is the smallest value, then `x-1` cannot be smaller. Note that this does not depend on the signed number representation; some machines may produce an exception on underflow, but, when x is the smallest value, none produces a value that will produce false for this comparison. However, gcc assumes that signed integer underflow does not happen, and uses this assumption to "optimize" this test to always produce false.[11]

So how do we rewrite this into "C"? If we assume that the type of x is `long`, we can write `x==LONG_MIN`.[12] Let us look at the code quality. For the `x-1>=x` variant, we use the gcc option `-fwrapv` that some (but not all) versions of gcc offer (as non-default option) to allow compiling many programs with signed integer overflows as intended. On AMD64 gcc-5.2.0 produces:

```
x-1>=x
48 8d 47 ff                   lea -0x1(%rdi),%rax
48 39 c7                      cmp %rax,%rdi
7f 06                         jg ...

x==LONG_MIN
48 b8 00 00 00 00 00 00 00 80 movabs $0x8000000000000000,%rax
48 39 c7                      cmp %rax,%rdi
75 05                         jne ...
```

Three instructions each, but the "C" variant takes 15 bytes (both with `-O3` and `-Os` (size optimization)), whereas the C* variant takes 9. Another way to implement this check would be

```
48 ff cf                      dec %rdi
71 05                         jno ...
```

This takes two instructions and 5 bytes. An optimizing compiler would ideally produce this machine code from all ways of expressing this check; that would be an optimization*, and a particularly useful one, because one cannot express this code more directly in C (the overflow flag is not a feature in the C programming language).

Another example[13] is the implementation of 32-bit rotation: The straightforward implementation in C*is `(x<<n)|(x>>32-n)` and compiles to the intended code in current compilers but performs undefined behavior in "C" when $n = 0$.[14] A "C" version with the obvious zero check generates additional instructions on both GCC and Clang. The blog entry also looks at another variant

[11] This even happens for `-O0` on, e.g., gcc-4.1.2 on Alpha.

[12] The type of x in the actual program may be a different signed integer type, resulting in much more code to handle all these possible types, while `x-1>=x` is independent of the type.

[13] From http://blog.regehr.org/archives/1063

[14] Presumably the reason for the undefined behaviour is that the machine instruction for `x>>32` produces x on some CPUs and 0 on others; note that either behavior produces the correct result for this idiom.

`(x<<n)|(x>>(-n&31))`,[15] which gcc managed to compile into a `rol`, but Clang didn't and produced code that was longer by four instructions.

So, converting C*code to "C" can lead to worse code even on those compilers whose maintainers say that we should do the conversion. You don't see this effect on benchmarks, because the benchmarks are not changed to eliminate undefined behaviour.

4 Source-level optimizations

Programmers can be very effective at improving the performance of code, and in particular, can do things that the compiler cannot do. Here we first look at three examples.

4.1 SPECint

Wang et al. [WCC+12] did not just present the difference that "optimizations" make for SPECint numbers (Section 2.1), they also looked at the reasons for these performance differences, and found two small source-level changes that made the less "optimizing" compiler variants produce just as fast code as the default compilers.

For 456.hmmer, the problem is that an `int` index is sign-extended in every iteration of an inner loop unless the sign-extension is "optimized" away by assuming that the int does not wrap around. The source-level solution is to use an address-length or unsigned type for the index; Wang et al. used `size_t`, and the slowdown from disabling "optimizations" went away for 456.hmmer.

For 462.libquantum, the problem is a loop-invariant load in the inner loop that could not be moved out of the loop without assuming strict aliasing, because there is a memory store (to a different type) in the loop. The source-level solution is to move that memory access out of the loop. Wang et al. did that, and the slowdown from disabling "optimizations" went away for 462.libquantum. Note that, if the store was to the same type as the load, "optimization" could not move the invariant load out, while source-level optimization still can.

Also note that you just need to look at the inner loops to find and perform such optimization opportunities, while you have to work through your whole program to convert it to "C", plus you may incur slowdowns from the conversion.

4.2 Jon Bentley's Traveling Salesman programs

In "Writing Efficient Programs" [Ben82], Jon Bentley used a relatively short traveling-salesman program that uses a greedy (non-optimal) algorithm as running example for demonstrating various optimizations at the source code level; each optimization step resulted in a new program. The programs in the book were written in Pascal. I transliterated them to C in 2001, keeping the original optimizations and a Pascal-like style (arrays instead of explicit pointer arithmetics),

[15] `n` needs to be unsigned in order for this variant to be "C".

except that I did not perform Bentley's step 7 of switching from floating point to integer arithmetics. This results in the programs `tsp1`...`tsp9` (but without `tsp7`).[16]

Given the Pascal-based style, I expect that these programs benefit from optimizations like strength reduction and induction variable elimination more than many other C programs. They are probably also closer to "C" than many other C programs for the same reason; to test this, I compiled these programs with `gcc-5.2.0 -m32 -fsanitize=undefined -fsanitize-undefined-trap-on-error`, and they ran through (but note that these runs may still perform undefined behaviors that the checker does not check, or for other inputs).

Experimental setup: We used several compiler versions and different options:

gcc We used gcc versions 2.7.2.3 (1997), egcs-1.1.2 (1999, shortly before gcc-2.95), and gcc-5.2.0 (2015). The earlier versions already "optimize" `x-1>=x` (with no way to disable this "optimization"), but overall probably perform much less "optimization" than gcc-5.2.0; e.g., egcs-1.1.2 has an option `-fstrict-aliasing`[17], but (unlike gcc-5.2.0) does not enable it by default, and gcc-2.7.2.3 does not even have such an option. In addition to "optimizations", hopefully gcc also added optimizations* in these 18 years. We use the `-m32` option for gcc-5.2.0 to produce IA-32 code, because the earlier compilers do not produce code for the AMD64 architecture that was only introduced in 2003.

Clang/LLVM We also use clang 3.5 (2014) for breadth of coverage. We use `-m32 -mno-sse` for comparability with the gcc results which also produce code for IA-32 without SSE.

`-fno...` In addition to using `-O3` and default optimizations (including "optimizations") we also compiled with `-O3` but disabled as many "optimizations" as we could identify: For gcc-5.2.0 we used `-fno-aggressive-loop-optimizations -fno-unsafe-loop-optimizations -fno-delete-null-pointer-checks -fno-strict-aliasing -fno-strict-overflow -fno-isolate-erroneous-paths-dereference -fwrapv`, for clang 3.5 we use `-fno-strict-aliasing -fno-strict-overflow -fwrapv`.[18] These compilers still might perform various "optimizations" that are not covered with these flags, but that is as close to turning these compilers into C* compilers as we can get.

`-O0` It is often suggested to turn off optimization in order to get a C* compiler. While this does not work (some gcc versions "optimize" `x-1>=x` even at `-O0`), we tried `-O0` to see how much it hurts performance.

[16] http://www.complang.tuwien.ac.at/anton/lvas/effizienz/tsp.html

[17] With strict aliasing the compiler assumes roughly that pointers to different types do not point to the same object.

[18] For clang I used the subset of the gcc options that clang accepts, because I could not find documentation on any such options.

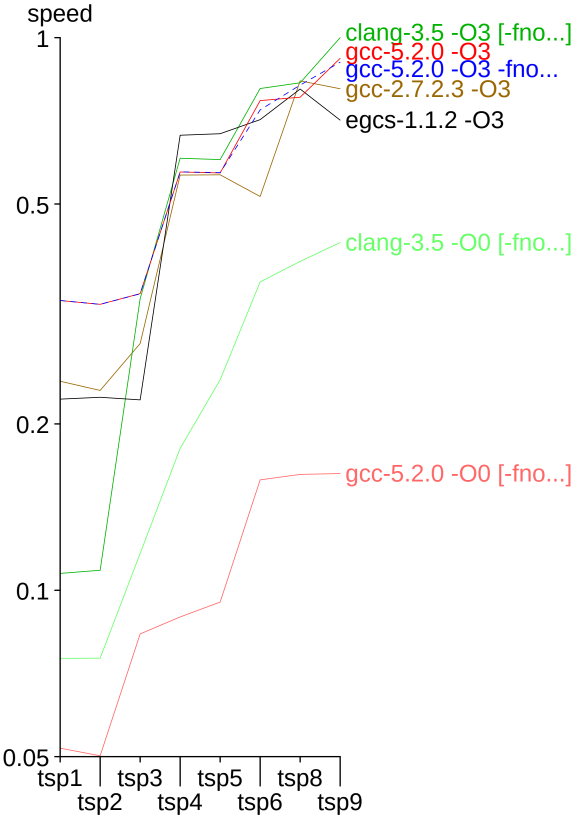


Fig. 1. Performance across different compilers of Jon Bentley's Traveling Salesman program variants for visiting 5000 cities

We ran the resulting binaries on a Core i3-3227U (Ivy Bridge), with 5000 cities. We measured the run-time in cycles using CPU performance counters with `perf stat -r 100 -e cycles`; this runs the program 100 times and reports "average" (probably arithmetic mean) and standard deviation; the standard deviation for our measurements was at most 0.62%.

Figure 1 shows the results. In cases where the binaries where the same with and without the `-fno...` options, one line is shown for both, with the label containing `[-fno...]`.

Source-level optimization speedup: Overall, the source-level optimizations provide a good speedup (starting at a factor 2.7 between tsp1 and tsp9 for `gcc-5.2.0 -O3`) across all compilers and options, with some optimization steps having little effect on performance, while others have a larger effect. This also disproves claims that compiler optimizers are now so good that they make source-level optimization unnecessary; on the contrary, compilers do not perform most of these source-level optimizations from a 33-year old book. If compilers performed these optimizations themselves, we would expect the `-O3` lines to be flat, but in fact the speedups from source-level optimizations provide similar speedups to the `-O3`-compiled versions as to the `-O0`-compiled versions; only the optimization from tsp4–tsp5 (inlining one function) is performed by the compilers with `-O3`, resulting in horizontal line segments for this step. Section 4.4 looks at why source-level optimization is effective.

“Optimization” speedup: We first compare the binaries generated with and without the `-fno...` options. For `clang-3.5 -O3`, `clang-3.5 -O0` and `gcc-5.2.0 -O0`, all of the programs are compiled to the same binary code without and with `-fno...` options. So for these compilers show only one set of results. For `gcc-5.2.0 -O3`, there are no differences in the binaries for tsp1...tsp3, but there are for tsp4...tsp9, so we measure both sets of binaries for `gcc-5.2.0 -O3`.

For tsp4/5 (and of course for tsp1–3, where there is no difference in the binaries) the code generated by `gcc-5.2.0 -O3` has the same speed as the code compiled with the `-fno...` options; for tsp6 it is faster by a factor of 1.04, for tsp8 it is *slower* by a factor of 1.05, and for tsp9 it is faster by a factor of 1.02. So disabling these “optimizations” has only a minor and inconsistent effect on performance.

`-O0` vs. `-O3`: By contrast disabling both “optimization” and optimization* with `-O0` has a dramatic effect on performance, especially for gcc (factor 5.6 for tsp9). So using `-O0` is not a good suggestion as a C* compiler if you care for performance: In addition to still performing some “optimizations”, it also produces slow code.

Other results: Every compiler has some program for which it is fastest: gcc-2.7.2.3 is fastest for tsp8, egcs-1.1.2 is fastest for tsp4/5, gcc-5.2.0 is fastest for tsp1–3, and clang-3.5 is fastest for tsp6 and tsp9. Overall, the performance with `-O3` is remarkably close given the 18 years span the gcc versions; Proebsting’s law tongue-in-cheekly predicts a factor of 2 between gcc-2.7.2.3 and 5.2.0.

The slow speed of `clang-3.5 -O3` for tsp1/2 is probably due to differences in `sqrt()` implementation, because tsp1/2 have a large number of calls to `sqrt()`, while that number is much smaller for tsp3–tsp9.

4.3 Gforth

Gforth is an implementation of the Forth programming language. Its virtual machine was implemented as a threaded-code interpreter starting in 1992 [Ert93]; already that version heavily used GNU C extensions to achieve better performance than is possible in standard C* (let alone “C”). And while it is extremely far from being a strictly conforming (and thus portable) program according to

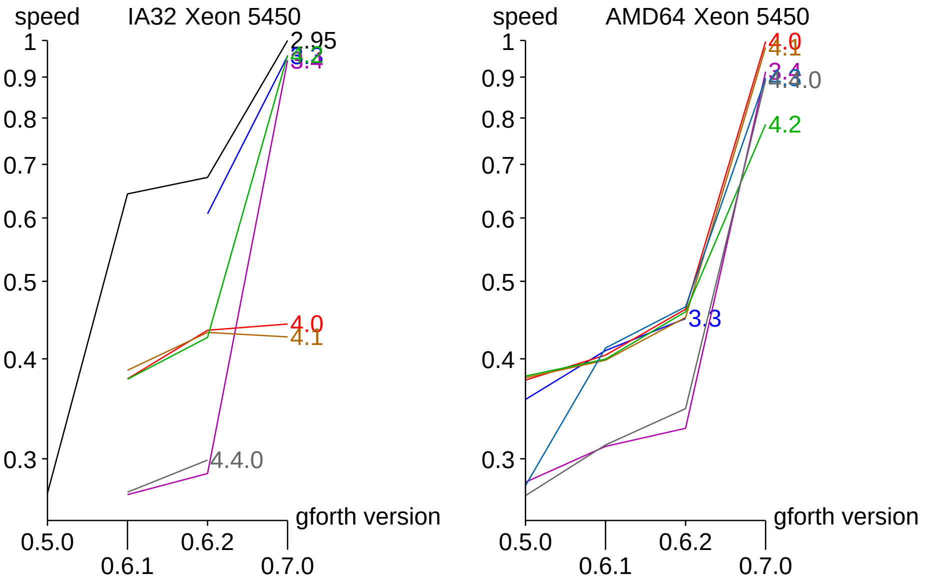


Fig. 2. Gforth performance with different GCC versions

the C standard, it is pretty portable: e.g., we tested Gforth 0.7.0 on eight different architectures, five operating systems, and up to nine gcc versions per architecture.

In 2009 we compared the performance of different Gforth versions across different CPUs and different gcc versions [Ert09], and below we discuss the results as relevant for the present paper. We measured different Gforth versions, from Gforth 0.5.0 (2000) to 0.7.0 (2008) compiled with various GCC versions from 2.95 (1999) to 4.4.0 (2009).

We ran five application benchmarks on a 3GHz Xeon E5450, each one three times per configuration, taking the median of the three runs and the geometric mean over these inividual benchmark results. The data we present here is the same as in Figures 8 and 9 of the Gforth performance paper [Ert09], but we present it differently.

Figure 2 shows 32-bit and 64-bit[19] results for different Gforth and GCC versions. As you can see, the source-level optimizations between Gforth 0.5.0 (2000) and 0.7.0 (2008) provide a speedup by a factor > 2.6 on most compiler versions. Some compiler versions don't perform as well as others, either because a source-level optimization was disabled[20], or because of some compiler-specific

[19] For AMD64, generic code is used up to 0.6.2, special AMD64 support was added only in 0.7.0.

[20] Gforth checks whether the assumptions used by the optimization hold, and falls back to older techniques if they do not.

problem (like bad register allocation: gcc-4.0 and -4.1 on IA32); for details, see the original paper [Ert09].

We also prototyped an optimization that moves Gforth further into JIT compiler territory, with less per-target effort than required by a conventional JIT compiler [EG04], and this optimization produced a median speedup of 1.32 on an Athlon, and 1.52 on a PPC 7400.

We had plans to turn this optimization into a production feature, but eventually realized that the GCC maintainers only care about certain benchmarks and treat production programmers as adversaries that deserve getting their code broken. Given that the Gforth source code is extremely far from "C", and this optimization would have taken it even further from "C", we dropped the plans for turning this feature into a production feature. So, in this instance, the focus on "C" and "optimizations" by the C compiler maintainers resulted in less performance overall.

The current GCC and Clang maintainers suggest that we should convert programs to "C". How would that turn out for Gforth?

We could not use explicit register allocation, and therefore lose much of the performance advantage of gforth-0.7.0. More importantly, we have to drop dynamic superinstructions, and, since everything else builds on that, that would throw us back to Gforth-0.5.0 performance. In addition, we would have to drop threaded code, so we would lose even more performance. We could get back a little performance by implementing static superinstructions and static stack caching in a new way (without dynamic superinstructions), but overall I expect a slowdown by a factor > 3 compared to gforth-0.7.0 from these changes alone.

Gforth is outside "C" in many other respects, in particular in its data and memory model. It is unclear to me how that part could be turned into efficient "C" code, but it certainly will not increase performance.

Changing the code to "C" would not just reduce performance by a lot, but also require a huge effort. Instead of spending that effort to make Gforth slower, we are considering switching to native code compilation, and getting rid of C as much as possible. Machine code, while not as portable as C used to be, offers the needed expressive power, reliability, and stability, that gcc used to give us, but no longer does.

4.4 Why are programmers effective?

Gcc-5.2.0 does not perform most of the optimizations that Bently performed in his 33 years old book. Why?

One big advantage that the programmers have is that they have to satisfy the requirements document (or the specification) of the program, while the compiler uses the source code as specification and has to keep to that. The source code overspecifies the program, so the compiler cannot perform the same optimizations that the programmer can. For example, tsp8 does not produce the same stdout output as tsp6, so a compiler could not perform that change.

Another advantage that programmers have is that they can have a better understanding of the design of the program, and therefore can perform transformations that the compiler cannot perform because it cannot determine that the transformation is safe or profitable to perform. As an example, in Gforth we have numbers for the VM instructions in the image file format, and replace them with code addresses when loading the image file, eliminating the need to do the code address lookup at run-time; compilers do not do that, because they cannot prove that the instruction numbers are not used in any other way.

Finally, programmers can apply optimizations for idioms for which compiler maintainers do not develop optimizations because they do not occur frequently enough in “relevant” code (i.e., their benchmarks).

Of course, source-level optimization costs in development, and may also cost in maintenance. However, the recommendation by “optimization” advocates of “fixing” or “sanitizing” your code (i.e., converting it to “C”) also costs in development and in maintenance.

As can be seen here, the optimization by programmers is far more effective than “optimization” by compilers, so it makes more sense to have a compiler with only optimization* and invest the development budget for optimization in source-level optimization rather than “sanitizing”. In particular, in source-level optimization you can concentrate on the parts of the programs relevant to performance, and stop when the benefit/cost ratio of further optimizations becomes too small, while “sanitizing” has to cover the whole program, as any undefined behavior in the program allows nasal demons, or “optimizations”, to happen.

5 Perspectives

Given the state of things, what should be done?

5.1 Compilers

Compiler maintainers should change their attitude: Programs that work with previous versions of the compiler on the same platform are not buggy just because they exhibit undefined behaviour; after all, they are conforming C programs. So the compiler should be conservative and disable “optimizations” by default, in particular new or enhanced “optimizations”. If you really want the default to be “C”, then at least provide a single, stable option for disabling “optimizations”; in this way, a C* program compiled with that option will also work with the next version of the compiler.

Also, there is still a lot of improvement possible in terms of optimizations*, and it would be a good idea to shift effort from “optimizations” to optimizations*.

I do not see a good reason for having “optimizations” at all, but if a compiler writer wants to implement them, it would be a good idea to be able to warn when “optimizations” actually have an effect. Contrary to what Chris Lattner claims[21], this is not that hard: Just keep track of both facts* and “facts”. When

[21] http://blog.llvm.org/2011/05/what-every-c-programmer-should-know_21.html

eventually generating code, if the code generated based on just facts* would be different from the code generated based on "facts", provide a warning. Providing a good explanation for the warning may be hard, but that's just an indication of how unintuitive the "optimization" is.

5.2 Standards

The compiler maintainers try to deflect from their responsibility for the situation by pointing at the C standard. But the C standard actually takes a very different position from what the compiler maintainers want to make us believe. In particular, the C99 rationale[22] [C03] states:

> **C code can be non-portable.** Although it strove to give programmers the opportunity to write truly portable programs, the C89 Committee did not want to force programmers into writing portably, to preclude the use of C as a "high-level assembler": the ability to write machine-specific code is one of the strengths of C. It is this principle which largely motivates drawing the distinction between strictly conforming program and conforming program (§4).
>
> **Keep the spirit of C.** The C89 Committee kept as a major goal to preserve the traditional spirit of C. There are many facets of the spirit of C, but the essence is a community sentiment of the underlying principles upon which the C language is based. Some of the facets of the spirit of C can be summarized in phrases like:
>
> – Trust the programmer.
> – Don't prevent the programmer from doing what needs to be done.
> – Keep the language small and simple.
> – Provide only one way to do an operation.
> – Make it fast, even if it is not guaranteed to be portable.
>
> The last proverb needs a little explanation. The potential for efficient code generation is one of the most important strengths of C. To help ensure that no code explosion occurs for what appears to be a very simple operation, many operations are defined to be how the target machine's hardware does it[23] rather than by a general abstract rule.

Still, the C standard blesses compilers like GCC and Clang that intentionally compile only strictly conforming C programs as intended as conforming implementations. There is the hope that market forces will drive compilers towards higher quality-of-implementation; unfortunately, that does not seem to work out: The compiler maintainers' perception of quality-of-implementation is based on benchmark results, and this perception has led to the current situation; and since both GCC and Clang maintainers take the same view of non-"C" programs, and

[22] The C11 rationale has not been published yet.

[23] By contrast, "what the hardware does" seems to be an insult in GCC circles: `news:<07qdndj_37WIDp7OnZ2dnUVZ_tmdnZ2d@supernews.com>`.

there is no competition from an optimizing* C* compiler in the free-software world, there is little that market forces can do in that area.

So, while the standard is not responsible for the current situation (the maintainers of these C compilers are), the C standards committee might still tighten the standard such that it cannot be abused by compiler maintainers in this way.

Instead of not specifying behavior (or explicitly specifying undefined behavior) for many cases where different C* implementations may produce different results, the standard should enumerate the possible results in enough detail to discourage "optimizing" compilers.

Pascal Cuoq, Matthew Flat and John Regehr suggested such an approach[24]: They want to define a friendly dialect of C by working through the list of undefined behaviours in the C11 standard, and reduce the amount of undefinedness, e.g., by replacing "has undefined behaviour" with "results in an unspecified value". This is definitely going in the right direction. There will still remain a gap between the way programmers think about the language construct and the way the friendly C specification describes it, but if the friendly C specification is tight enough to rule out "optimizations", the remaining gap may still be a source of joy for language lawyers, but harmless for practical programmers. "Unspecified value" may not be quite tight enough, though, because it does not give a specified result for, e.g., `x-x` when x is uninitialized.

5.3 Programmers

The attitude of compiler maintainers puts programmers in a tough situation. The next version of the compiler could break their currently-working program. The simplest way to deal with that is to stick with the compiler version that works with your program. You may need to keep the old binaries, or compile the old sources with a newer compiler (I have built gcc-2.7.2.3 with gcc-4.8 for this paper).

If you want to make your program work for newer versions of the compiler (e.g., because you want to port to an architecture that is not supported by the old version), you can use the flags that define some of the undefined behaviours (such as those listed for our experiments in Section 4.2). You can also see if lowering the optimization level helps.

As a long-term perspective, you could also decide to switch to another programming language. Unfortunately, there are not that many languages available that occupy the C* niche: a low-level language that can be used as portable replacement for assembly language; in particular, C ate all its brethren in the Algol family (e.g., Bliss, BCPL). Forth is available, but is probably unattractive to most programmers coming from Algol-like languages. C-- [JRR99] was intended as a portable assembly language, but seems to have been relegated to an internal component of the Glasgow Haskell Compiler.

Instead of using a portable assembly language, you can go for real assembly language; in earlier times C was less cumbersome and therefore more attractive,

[24] http://blog.regehr.org/archives/1180

but that has changed. In contrast to C as currently implemented, assembly language is rock-solid. The disadvantage of assembly language is that it is not portable, but the number of different architectures in general-purpose computers has fallen a lot since the 1990s, so you may be able to make do with just one or two.

You probably don't want to write everything in assembly language, but most code in higher-level languages. There are a number of newer Algol-family languages that are slightly higher-level than C used to be that may be appropriate for the higher-level language part, e.g., Rust, D, and Go. I do not have experience with these languages, nor have I examined the specification and the attitude of the compiler maintainers, so unfortunately I cannot make any recommendations here.

Our perspective for Gforth is to use Forth as high-level language and assembly/machine language for the parts that cannot be done in Forth (for these parts we currently use GNU C).

5.4 Tools and Research Opportunities

Just as the existence of "C" compilers has spawned tools and research papers on finding undefined behaviors and code that may be "optimized" away although it should not, a focus on $C^\star$ compilers and source-level optimization could lead to tools for finding source-level optimization opportunities.

E.g., consider the source-level optimizations Wang et al. [WCC+12] used on SPECint: To get rid of sign extensions, a tool could perform run-time checks to see if using `long` instead of `int` produces a different result, and if not, could then suggest to the programmer to change the type of inner-loop induction variables accordingly. A tool could also check whether a load in an inner loop always produces the same result during a run, and, if so, suggest to the programmer to move the load out of the loop manually; the source-level optimization also works in cases where "optimization" does not work because there is a store in the loop to the same type as the load.

The research questions are what other optimizations can be supported by such tools, and how effective they are. Of course, the possible optimizations are not limited to those that are possible by converting to "C" and enabling "optimizations".

6 Related work

Complications generate interesting research questions, even if they are unnecessary complications.

Wang et al. [WCC+12] describe a number of "optimizations" by "C" compilers, and give examples where the "optimizations" were not optimizations, but led to $C^\star$ code not being compiled as intended; most of the examples are for security vulnerabilities caused by "optimizations". It also gives performance results showing that "optimizations" give a very small speedup even for SPECint.

There is work on finding undefined behavior with run-time checks, e.g., for integer computations [DLRA12]. Of particular interest for the current work are the empirical results of applying these checks to various programs, in particular, how widespread these undefined behaviors are.

Not all undefined behaviors can be exploited by "optimizations", and another paper by Wang et al. [WZKSL13] describes a tool for finding code that may be "optimized" (but not optimized*) away. In addition, they describe a number of security vulnerabilities caused by "optimizations" and also gives empirical results on how many program fragments are "optimized" away in how many programs.

7 Conclusion

"Optimizations" based on assuming that undefined behavior does not happen buy little performance even for SPECint benchmarks (1.7% speedup with Clang-3.1, 1.1% with GCC-4.7), and incur large costs: Most production programs perform undefined behaviors and removing them all requires a lot of effort, and may cause worse code to be produced for the program. Moreover, a number of security checks have been "optimized" away, leaving the affected programs vulnerable.

If you are prepared to invest that much effort in your program for performance, it is much better to invest it directly in source-level optimization instead of in removing undefined behavior. E.g., just two small source-level changes give the same speedups for SPECint as the "optimizations"; we also presented examples of source-level optimizations that buy speedup factors > 2.6.

Compiler writers should disable "optimizations" by default, or should at least give the programmers a single flag to disable them all and that also disables new "optimizations" in future compiler versions. A focus on optimizations* and on supporting source-level optimizations better would also be welcome. Finally, programs that work on a version of your compiler are conforming C programs and they are not buggy just because they perform undefined behavior.

References

Bac81. John Backus. The history of Fortran I, II, and III. In Richard L. Wexelblat, editor, *History of Programming Languages*, pages 25–45. Academic Press, 1981. 1

Ben82. Jon L. Bentley. *Writing Efficient Programs*. Prentice-Hall, 1982. 4.2

C03. *Rationale for International Standard—Programming Languages—C*, revision 5.10 edition, 2003. 5.2

DLRA12. Will Dietz, Peng Li, John Regehr, and Vikram Adve. Understanding integer overflow in C/C++. In *34th International Conference on Software Engineering (ICSE)*, 2012. 2, 6

EG04. M. Anton Ertl and David Gregg. Retargeting JIT compilers by using C-compiler generated executable code. In *Parallel Architecture and Compilation Techniques (PACT' 04)*, pages 41–50, 2004. 4.3

Ert93. M. Anton Ertl. A portable Forth engine. In *EuroFORTH '93 conference proceedings*, Mariánské Lázně (Marienbad), 1993. 4.3

Ert09. M. Anton Ertl. A look at Gforth performance. In *25th EuroForth Conference*, pages 23–31, 2009. 4.3

JRR99. Simon L. Peyton Jones, Norman Ramsey, and Fermin Reig. C--: a portable assembly language that supports garbage collection. In *International Conference on Principles and Practice of Declarative Programming*, September 1999. 5.3

WCC$^+$12. Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Undefined behavior: What happened to my code? In *Asia-Pacific Workshop on Systems (APSYS'12)*, 2012. 2.1, 2.3, 4.1, 5.4, 6

WZKSL13. Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: analyzing the impact of undefined behavior. In Michael Kaminsky and Mike Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 260–275. ACM, 2013. 2.1, 2.3, 6

Checking Spring Annotations

Konrad Fögen, Vincent von Hof, and Herbert Kuchen

University of Münster, IS Department, Leonardo Campus 3, D-48149 Münster,
konrad.foegen@uni-muenster.de, von.hof@wi.uni-muenster.de,
kuchen@uni-muenster.de,
WWW: https://www.wi.uni-muenster.de/department/groups/pi/

Abstract. Dependency injection frameworks such as the Spring framework rely on dynamic language features of Java. Errors arising from the improper usage of these features bypass the compile-time checks of the Java compiler. This paper discusses the application of static code analysis as a means to restore compile-time checking for Spring-related configuration errors. First, possible errors in the configuration of Spring are identified and classified. Attributed grammars are applied in order to formally detect the errors and a prototypical compiler extension is implemented based on Java's pluggable annotation processing API.

1 Introduction

The Java programming language is one of the most popular programming languages in general and especially in the field of enterprise applications[1] [Wal14, p.3]. In addition, *dependency injection* (DI) is frequently used to support and simplify the development of Java applications. DI is a *creational design pattern* that abstracts away the process of object creation and composition [GHJV95, p.94]. Typical implementations are generic and make no assumptions about the objects they manage. Instead, they rely on an external configuration [Pra09, p.17].

However, the generic implementation requires the use of dynamic language features such as the *Java Reflection API* [Ora15]. Despite its necessity and usefulness, the Java Reflection API has a downside, since errors arising from the improper application cannot be detected by available Java compilers. Thereby, the detection of errors is shifted from compile-time to runtime.

Since the errors are not automatically detected at compile-time and the manual detection is tedious, developers have a particular interest in automatic solutions to detect them as early as possible, preferably at compile-time during the development.

There are quite a few tools for the *static analysis* of Java programs such as FindBugs [APM$^+$07], Checkstyle [Bur03], PMD [PMD15], SonarQube

[1] See http://www.langpop.com or http://lang-index.sourceforge.net/ as indicators for Java's popularity.

		Spring Context	
		Dependent	Independent
Analysis Level	Above Method Level	Group I (13 errors)	Group II (13 errors)
	Below Method Level	Group III (8 errors)	Group IV (4 errors)

Table 1. Classification of Spring Configuration Errors

[Son15], Java Language Extender [VWKBJ06], ESC/Java2 [CK05], JastAddJ [EH07], JavaCOP [MME$^+$10], JQual [GF07], and the Checker framework [PAC$^+$08]. All these tools are general purpose inspection tools. To the best of our knowledge, there is no tool for the static detection of Spring configuration errors.

Our approach is based on attributed grammars [Knu68] and a compiler extension to detect errors arising from the improper application of the *Spring framework* [Piv15] at compile-time. Spring is chosen as a representative for the multitude of different DI implementations for Java, since it is one of the most popular implementations. Furthermore, its configuration is based on Java annotations which makes it very suitable for pluggable annotation processing.

This paper is structured as follows. In Section 2, Spring configuration errors are identified and classified. In Section 3, attributed grammars are provided which formally describe the detection of Spring configuration errors. Using these attributed grammars, a prototypical compiler extension based on Java's pluggable annotation processing API is described in Section 4. The insights gained by the development of the prototype are used to evaluate the approach in Section 5. In Section 6, we conclude and point out future work.

2 Spring Configuration Errors

The Spring framework is an open source framework which implements the dependency injection design pattern. At its core, the *Spring context* component provides its clients with requested and dependent objects, so-called *beans* which can be any kind of simple, Plain Old Java Objects (POJOs) [Wal14, p.4].

Several different context implementations exist which mainly differ in terms of their configuration format, e.g. Java-based or XML-based configurations. In general, configurations consist of features referring to the actual *Spring context* as well as to a set of Spring bean definitions.[2] Besides that, three different ways of defining *Spring beans* are supported by Spring: explicit configurations via Java and XML as well as implicit configurations via Java annotations. For more information about Spring, please refer to [Wal14], [Pra09] or [Joh15].

A literature review of the Spring framework reference [Joh15] and expert interviews have been conducted in order to identify different types of errors. As a result, 38 distinct error types have been identified and classified into four groups as depicted in Table 1. In the present paper, the core container and the

[2] This work focuses only on Java-based configurations.

data access / integration modules are considered, since they can be used by any Spring-based Java application. The classification depends on the core features of the Spring framework and the usage of these features determines the assignment of an error to a certain dimension. Four groups are formed by two dimensions each featuring two manifestations. The spring context dimension determines whether or not the analysis requires information derived from the Spring context. Analysis level is the second dimension and determines whether or not the analysis requires information about the control flow that concerns language constructs below the method-level, e.g. method invocations or variable assignments. In the following, exemplary errors are described for further illustration of the different error types.

Group I. Errors belonging to this group depend on the Spring context and another component which uses a related Spring-specific annotation. The errors occur *above* the method level, e.g. declarations of classes, methods or member variables. Viewed in isolation, the Spring context configuration and the component may not be erroneous but their interaction is.

As an example, Spring's transaction infrastructure encapsulates the internals of specific transaction management APIs and offers a declarative model for the integration into applications [Joh15, chap.12.3]. A Spring context which defines a transaction manager as well as a *@EnableTransactionManagement* annotation are required to enable the transaction management. Once enabled, the *@Transactional* annotation can be attached to methods in order to enable transactional support for the method. Listing 1.1 illustrates the correct usage.

```
 1  @Configuration
 2  @EnableTransactionManagement
 3  public class SpringConfig {
 4    @Bean
 5    public PlatformTransactionManager transactionmanager() {
          ...}
 6  }
 7
 8  @Component
 9  public class PrinterService {
10    @Transactional
11    public void print() {...}
12  }
```

Listing 1.1. Illustration of Spring's Transaction Management

In this situation, errors occur if transaction management is enabled but not used because no methods are annotated with *@Transactional*, i.e. when line 10 is removed from the listing. Or - the opposite situation - if *@Transactional* methods exist but the transaction management is not enabled, i.e. a situation where line 2 is removed from the listing.

Group II. Errors in group II are Spring context-independent and occur above the method level. They consist of annotated language constructs which - at the same time - have some other attributes or other annotations which are incompatible to that original annotation.

For instance, the Spring framework uses the *@Autowired* annotation to mark methods or fields for annotation-based injection [Wal14, p.39]. Sometimes, dependencies are ambiguous and the Spring context finds several bean definitions that match and then has to choose between them [Wal14, p.75]. The *@Qualifier* annotation can be used in conjunction with *@Autowired* to narrow the result set. It is an error to use *@Qualifier* without a corresponding *@Autowired* annotation.

The *@Qualifier* annotation can also be used indirectly. There is no difference between the usage of *@Qualifier* and the usage of annotation types annotated with *@Qualifier*. Both, the error and the indirect usage of *@Qualifier* are illustrated in Listing 1.2.

```java
@Qualifier
@interface DinA4Format {...}

@Component
@DinA4Format
class DinA4DocumentFormatter implements DocumentFormatter {...}

@Component
class PrinterService {
  //@Autowired is missing
  @DinA4Format
  public PrinterService(DocFormatter f) {...}
}
```

Listing 1.2. Illustration of @Qualifier Without @Autowired

Group III. Similar to errors in group I, the errors in this group also depend on specific Spring context configurations. But in contrast, they also depend on language constructs *below* the method level. For instance, the lifecycle of a bean is an important aspect described by its *bean definition*. The lifecycle of a bean starts after the corresponding Spring context is initialized and ends right before the Spring context shuts down. In between that timeframe, the lifecycle of a bean is defined by its *scope* [Wal14, p.81]. The *singleton* scope is the default scope for beans, where only one shared instance of the bean exists per Spring context and it exists until the context shuts down [Joh15, chap.5.5.1]. In contrast, beans defined with the *prototype* scope are created as new instances every time they are requested.

One important aspect regarding prototype scope is that the Spring context does not manage the complete lifecycle of these beans. Even though prototype and singleton beans are initialized the same way, their destruction is different, since the Spring context does not store references to prototyped beans and therefore cannot initiate their destruction. A Spring bean definition is explicitly defined by attaching the *@Bean* annotation to a method or implicitly defined by adding the *@Component* annotation to a class declaration.

Furthermore, Spring allows to define *lifecycle callbacks*, e.g. methods called by the Spring context when a bean is constructed or destructed [Wal14, p.33]. Among other ways, they can be defined by annotating a corresponding method with *@PostConstruct* or *@PreDestroy*.

Since the Spring context cannot initiate the destruction process of prototyped beans, methods that qualify as destruction lifecycle callbacks are considered erroneous, if the beans are defined with the prototype scope. Listing 1.3 illustrates this error where the component contains a *close* method to cleanup resources which is never invoked due to the prototype scope.

```java
@Configuration
public class SpringConfig {
  @Bean
  @Scope(”prototype”)
  public PrinterService printerService() {
    return new PrinterService();
  }
}

public class PrinterService {
  @PreDestroy
  public void close() {   // will not be invoked
    this.usbConnection.close();
  }
}
```

Listing 1.3. Illustration of Callbacks on Prototyped Beans

Group IV. This group also comprises errors that occur below the method level and do not depend on the Spring context. An example is related to Spring's *JdbcTemplate* component which provides an abstraction layer covering specific details of Java's JDBC API. Here, SQL is used to interact with databases. The corresponding code is *linguistically separated* from the surrounding code written in Java. Therefore, the compiler cannot check whether or not SQL code is compliant to the language definition of SQL. A typical use case in this context is a developer who creates and tests SQL statements in a database tool. Once the SQL statement is ready, he copies it into a Java String and uses it. Statements which are executed in a database tool often require to be terminated with a semicolon. However, having that semicolon at the end of a SQL String in Java results in a runtime exception.

These kinds of problems can be detected by applying pluggable type systems similar to the one for regular expressions by the Checker framework [SDE12]. Therefore, such errors are not further discussed in this paper.

3 Error Detection via Attributed Grammars

Static code analysis is an analytical approach to detect lexical, syntactic, and also some semantic errors. The source code of software is analyzed in order to understand its structure (syntax) and meaning (semantics) [ALSU07, p.21]. The gained understanding is then used to identify errors within the source code. Regular expressions and context-free grammars can be used to define the lexical and syntactic structure of a programming language and errors can be detected by identifying mismatches between the defined structure and the actual source code.

Beyond syntactical checks, compilers may also check for the static semantics, e.g. whether a method is invoked with the right number and types of arguments. Or, in our case, that Spring configurations and annotations are used in a correct manner.

In 1968, KNUTH introduced *attributed grammars* as a formal approach to express and handle semantical aspects of a programming language [Knu68]. Attributed grammars are context-free grammars extended with attributes and semantic rules [SK95, pp.66-67]. Each nonterminal of the context-free grammar may have several attributes. Each attribute can either be *synthesized* or *inherited* and it has a value which is defined by a semantic rule associated with a production of the context-free grammar.

Roughly, if $A ::= B_1 \dots B_n$ (for $n \in \mathbb{N}$) is a context-free rule with nonterminals $A, B_1, \dots, B_n$, all of which have a synthesized attribute s and an inherited attribute i, then corresponding semantic rules can define the values of the attributes $A.s, B_1.i, \dots, B_n.i$ as follows:

$$A.s \leftarrow f(A.i, B_1.s, \dots, B_n.s) \tag{1}$$

$$B_j.i \leftarrow g(A.i, B_1.s, \dots, B_{j-1}.s, B_{j+1}.s, \dots, B_n.s) \tag{2}$$

where f and g are functions mapping attribute values to another attribute value and $j \in \{1, \dots, n\}$. If the context-free rules contain terminal symbols and / or nonterminals have several synthesized and inherited attributes, the formulas (1) and (2) have to be generalized accordingly (see [ALSU07] for a full description of attributed grammars).

The following notation is used within this paper to represent attributes, semantic rules, and conditions. Semantic rules are enclosed by curly brackets and are placed behind the body of the corresponding production. $\$0.a$ is used to refer to attribute a of the symbol on the left hand side (lhs) of the production, $\$1.a$ is used to refer to the leftmost symbol of the right hand side (rhs) and so forth. A reversed arrow ← is used to represent the value assignment from the value on the rhs of a semantic rule to the attribute on the lhs. For the rhs, we use a syntax similar to that of C or Java.

After constructing a syntax tree (via lexical and syntactic analysis), the attribute values of the symbols in that tree can be determined by applying the semantic rules [ALSU07, p.54]. The order in which the attributes can be evaluated has to reflect the dependencies of the attributes caused by the semantic rules. In general, there is no guarantee that an order exists in which all attributes of all nodes can be evaluated. Though, there are subclasses of attributed grammars which restrict the usage of attributes and semantic rules to guarantee the existence of an evaluation order [ALSU07, p.313].

```
⟨root⟩ ::= ⟨typedecllist⟩ | $

⟨typedecllist⟩ ::= ⟨typedecl⟩ ⟨typedecllist⟩ | ε

⟨typedecl⟩ ::= ⟨modifiers⟩ ⟨typedecltype⟩

⟨modifiers⟩ ::= ‘public’ ⟨modifiers⟩ | ‘private’ ⟨modifiers⟩
  |  ⟨annotation⟩ ⟨modifiers⟩ | ε

⟨annotation⟩ ::= ‘@’ ⟨identifier⟩ ⟨annoarguments⟩

⟨typedecltype⟩ ::= ⟨annotypedecl⟩ | ⟨classdecl⟩ | ⟨interfacedecl⟩

⟨annotypedecl⟩ ::= ‘@interface’ ⟨identifier⟩ ‘{’ ⟨annotypebody⟩ ‘}’

⟨classdecl⟩ ::= ‘class’ ⟨identifier⟩ ⟨superclass⟩ ⟨interfaces⟩ ‘{’ ⟨classbody⟩ ‘}’

⟨interfacedecl⟩ ::= ‘interface’ ⟨identifier⟩ ⟨superinterface⟩ ‘{’ ⟨interfacebody⟩ ‘}’

⟨classbody⟩ ::= ⟨modifiers⟩ ⟨type⟩ ⟨identifier⟩ ⟨classbodytype⟩ ⟨classbody⟩ | ε
```

Fig. 1. Java Grammar in BNF Notation (Excerpt).

Two subclasses relevant for this work are *S-* and *L-attributed grammars*: S-attributed grammars are grammars that only contain synthesized attributes and no inherited attributes [ALSU07, p.313]. They allow a bottom-up evaluation of attributes. L-attributed grammars also guarantee the existence of an evaluation order. Roughly, they allow the evaluation of attributes bottom up and left to right. See [ALSU07] for details.

We use an LL(1)-compliant context-free grammar which describes the subset of Java language constructs relevant for the detection of Spring configuration errors. Figure 1 provides an overview of the productions which are relevant for the succeeding analyses.

The semantic rules used for attributed grammars are based on the following constants and operations: *error* is used to indicate that an error has been detected, *emptySet* creates an empty set, *newSet* creates a set containing a single element, *intersects* returns true if and only if an intersection of two sets is not empty and *union* computes the union of two sets. The function *value* operates on identifiers and returns the actual value as a string.

In the following, an exemplary attributed grammars is provided which allow to expose an error explained in of Section 2.

Spring context-dependent errors that occur above the method level can be generically depend on the presence or absence of annotations. The presence or absence can be described via two synthesized boolean attributes *enabled* (1) and *used* (2). A boolean expression using the two attributes describes whether an error is present or not.

Consider again the error introduced in Listing 1.1. It can be exposed as follows. The attribute *enabled* is set to true, if a Spring configuration exists and if this configuration is annotated with *@EnableTransactionManagement*. The

Nonterminal Symbols	**Synthesized Attributes**
$\langle root \rangle$	-
$\langle typedecllist \rangle$	enabled, used
$\langle typedecl \rangle$	enabled, used
$\langle typedecltype \rangle$	used
$\langle classdecl \rangle$	used
$\langle classbody \rangle$	used
$\langle interfacedecl \rangle$	used
$\langle interfacebody \rangle$	used
$\langle modifiers \rangle$	names
$\langle annotation \rangle$	name

Table 2. Overview of Nonterminals and Synthesized Attributes.

attribute *used* is set to true, if a method annotated with *@Transactional* exists. Otherwise, the attributes are set to false. The error occurs if the attribute *enabled* is true and at the same time the attribute *used* is false. The condition can thus be expressed as the boolean expression $enabled \land \neg used$.

The detection can be described by an S-attributed grammar with four synthesized attributes *enabled*, *used*, *name* and *names* whereby *enabled* refers to occurrences of *@EnableTransactionManagement* and *used* refers to occurrences of *@Transactional. name* refers to the identifier of an annotation and *names* is a set of a names.

In the following, the semantic rules and the synthesis are described in greater detail. Table 2 provides an overview of the relevant nonterminals and their synthesized attributes. Figure 2 shows a corresponding S-attributed grammar. The synthesis starts with the collection of $\langle annotation \rangle$ names. $\langle annotation \rangle$ elements delegate their *name* to the enclosing $\langle modifiers \rangle$ element which collects them. For $\langle classbody \rangle$ and $\langle interfacebody \rangle$, the *used* attribute is set to true if the collected set of modifiers contains the *@Transactional* annotation.

The value of the *used* attribute is then propagated to the $\langle type \rangle$ declaration. The *used* attribute value for annotation types is always false since they cannot use the *@Transactional* semantics. The $\langle typedecl \rangle$ propagates the *used* value to the enclosing $\langle typedecllist \rangle$. In addition, it is checked whether the type declaration itself uses the *@Transactional* annotation. Besides that, it is checked whether the type declaration is the actual Spring context configuration class and if so, whether the transaction management is enabled or not. The *enabled* attribute represents the value of that check.

The semantic rules of $\langle typedecllist \rangle$ collect the attributes of each enclosed type declaration. The attributes are set to true if they are true for at least one type declaration. The attributes are then validated at the root production. An error is detected, if the transaction management is enabled but no method uses the *@Transactional* annotation.

Consider again Listing 1.1 where an error occurs if the transaction management is enabled but not used because no methods are annotated with *@Trans-*

$$\langle \mathit{annotation} \rangle ::= \text{`}\texttt{@}\text{'}\ \langle \mathit{identifier} \rangle \qquad \{\$0.\mathit{name} \leftarrow \mathit{value}(\$1);\}$$

$$\begin{array}{ll} \langle \mathit{modifiers} \rangle ::= \text{`}\texttt{public}\text{'}\ \langle \mathit{modifiers} \rangle & \{\$0.\mathit{names} \leftarrow \$1.\mathit{names};\} \\ \quad \mid\ \text{`}\texttt{private}\text{'}\ \langle \mathit{modifiers} \rangle & \{\$0.\mathit{names} \leftarrow \$1.\mathit{names};\} \\ \quad \mid\ \langle \mathit{annotation} \rangle\ \langle \mathit{modifiers} \rangle & \{\$0.\mathit{names} \leftarrow \mathit{union}(\mathit{newSet}(\$1.\mathit{name}), \$2.\mathit{names});\} \\ \quad \mid\ \varepsilon & \{\$0.\mathit{names} \leftarrow \mathit{emptySet}();\} \end{array}$$

$$\begin{array}{ll} \langle \mathit{classbody} \rangle ::= \langle \mathit{modifiers} \rangle\ \langle \mathit{type} \rangle\ \langle \mathit{identifier} \rangle\ \langle \mathit{classbodytype} \rangle\ \langle \mathit{classbody} \rangle & \\ \qquad \{\$0.\mathit{used} \leftarrow \mathit{intersects}(\mathit{newSet}(\text{`}\texttt{Transactional}\text{'}), \$1.\mathit{names});\} & \\ \quad \mid\ \varepsilon & \{\$0.\mathit{used} \leftarrow \mathit{false};\} \end{array}$$

$$\begin{array}{ll} \langle \mathit{interfacebody} \rangle ::= \langle \mathit{modifiers} \rangle\ \langle \mathit{type} \rangle\ \langle \mathit{identifier} \rangle\ \langle \mathit{methoddecl} \rangle\ \langle \mathit{interfacebody} \rangle & \\ \qquad \{\$0.\mathit{used} \leftarrow \mathit{intersects}(\mathit{newSet}(\text{`}\texttt{Transactional}\text{'}), \$1.\mathit{names});\} & \\ \quad \mid\ \varepsilon & \{\$0.\mathit{used} \leftarrow \mathit{false};\} \end{array}$$

$$\begin{array}{l} \langle \mathit{classdecl} \rangle ::= \text{`}\texttt{class}\text{'}\ \langle \mathit{identifier} \rangle\ \langle \mathit{superclass} \rangle\ \langle \mathit{interfaces} \rangle\ \text{`\{'}\ \langle \mathit{classbody} \rangle\ \text{`\}'} \\ \qquad \{\$0.\mathit{used} \leftarrow \$4.\mathit{used};\} \end{array}$$

$$\begin{array}{l} \langle \mathit{interfacedecl} \rangle ::= \text{`}\texttt{interface}\text{'}\ \langle \mathit{identifier} \rangle\ \langle \mathit{superinterface} \rangle\ \text{`\{'}\ \langle \mathit{interfacebody} \rangle\ \text{`\}'} \\ \qquad \{\$0.\mathit{used} \leftarrow \$3.\mathit{used};\} \end{array}$$

$$\begin{array}{ll} \langle \mathit{typedecltype} \rangle ::= \langle \mathit{annotypedecl} \rangle & \{\$0.\mathit{used} \leftarrow \mathit{false};\} \\ \quad \mid\ \langle \mathit{classdecl} \rangle & \{\$0.\mathit{used} \leftarrow \$1.\mathit{used};\} \\ \quad \mid\ \langle \mathit{interfacedecl} \rangle & \{\$0.\mathit{used} \leftarrow \$1.\mathit{used};\} \end{array}$$

$$\begin{array}{l} \langle \mathit{typedecl} \rangle ::= \langle \mathit{modifiers} \rangle\ \langle \mathit{typedecltype} \rangle \\ \qquad \{\$0.\mathit{used} \leftarrow \$2.\mathit{used} \parallel \mathit{intersects}(\mathit{newSet}(\text{`}\texttt{Transactional}\text{'}), \$1.\mathit{names}); \\ \qquad \$0.\mathit{enabled} \leftarrow \mathit{intersects}(\mathit{newSet}(\text{`}\texttt{Configuration}\text{'}), \$1.\mathit{names}) \\ \qquad\qquad \&\&\ \mathit{intersects}(\mathit{newSet}(\text{`}\texttt{EnableTransactionManagement}\text{'}), \$1.\mathit{names});\} \end{array}$$

$$\begin{array}{ll} \langle \mathit{typedecllist} \rangle ::= \langle \mathit{typedecl} \rangle\ \langle \mathit{typedecllist} \rangle & \\ \qquad \{\$0.\mathit{enabled} \leftarrow \$1.\mathit{enabled} \parallel \$2.\mathit{enabled};\quad \$0.\mathit{used} \leftarrow \$1.\mathit{used} \parallel \$2.\mathit{used};\} & \\ \quad \mid\ \varepsilon & \{\$0.\mathit{enabled} \leftarrow \mathit{false}; \$0.\mathit{used} \leftarrow \mathit{false};\} \end{array}$$

$$\langle \mathit{root} \rangle ::= \langle \mathit{typedecllist} \rangle \qquad \{\mathit{if}(\$1.\mathit{enabled}\ \&\&\ !\$1.\mathit{used})\{\mathit{error}();\}\} \quad \mid \quad \$$$

Fig. 2. S-attributed Grammar for detection of `Transactional` error (excerpt).

actional, i.e. when line 10 is removed from the listing. An excerpt of the abstract syntax tree (AST) annotated with the results from applying the before-discussed attributed grammar to Listing 1.1 is provided in Figure 3. The rounded rectangles represent attributes and their values belonging to a node and the dotted lines illustrate the bottom-up flow of the computation. The excerpt depicts the evaluation of the *SpringConfig* class declaration where the value of the *used* attribute is false since the declaration does not contain any methods annotated with *@Transactional*. However, the value of the *enabled* attribute is true because the class declaration is a Spring configuration and the transaction management is enabled.

Other errors above the method level can be detected by similar attributed grammars. For errors in group II, we use L-attributed grammars.

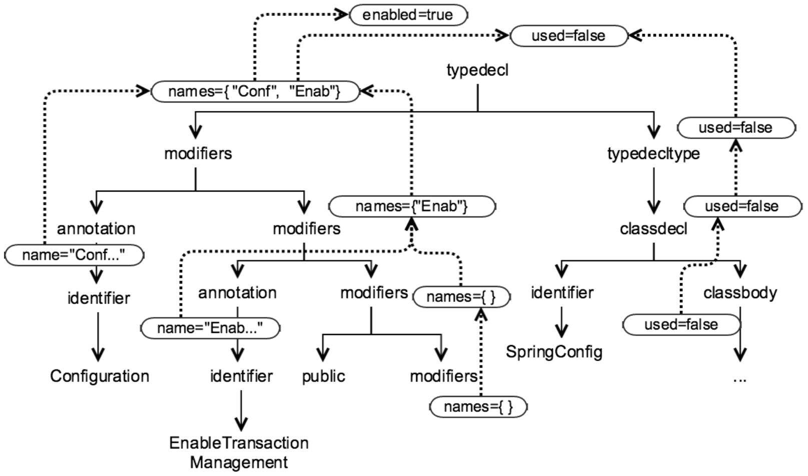


Fig. 3. Annotated parse tree for SpringConfig (excerpt).

For errors that occur *below* the method level, we transform the abstract syntax tree to a *control-flow graph* (CFG) [ALSU07], which is more suitable for dealing with the dynamic behavior of the code. On the CFG, we then perform a *reaching definitions* analysis [Muc97, p.218].

Using this analysis, we can e.g. detect the error regarding destruction methods of prototype-scoped beans as presented in Listing 1.3.

4 Prototypical Implementation

Our approach is based on the attributed grammars explained above. Technically, the corresponding analysis has been implemented using the *pluggable annotation processing API*. This API is specified by Java Specification Request (JSR) 269 and enables the processing of annotations at compile-time [Dar06]. It defines a language model representing the source code of a processed Java project. The design of the language model can be described by the composite design pattern ([GHJV95, p.183]).

Besides that, the API defines how compiler extensions can be declared and executed. Figure 4 illustrates the architecture of the Java compiler. It is based on a pipe and filter architecture as described in [LL12, p.432]. The extension depicts how our prototype is plugged into the compilation process via the pluggable annotation processing API. Once the compiler finishes the lexical and syntactical analysis of the source code, it invokes the prototype through the plugin interface. The *SpringAnnotationProcessor* component uses the aforementioned composite structure provided by the compiler to perform analyses based on the attributed grammars as described in Section 3.

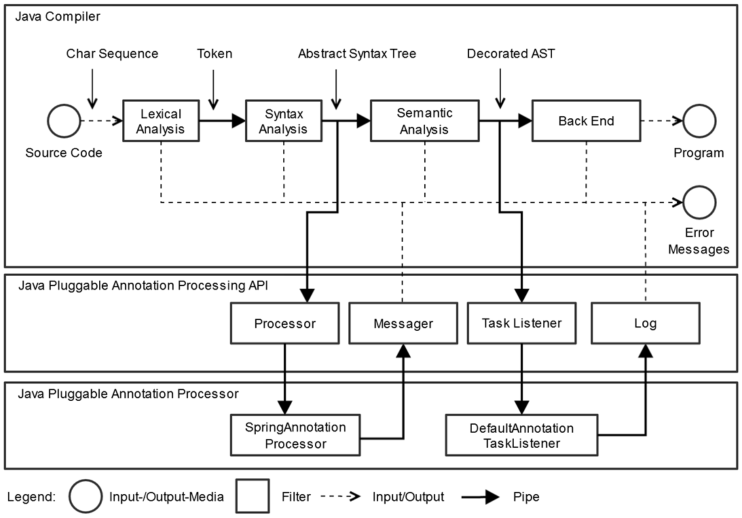


Fig. 4. Pipe and Filter Architecture of the Java Compiler

Detected errors are delegated to the *Messager* component which adds them to the compilers error messages. Next, the semantic analysis of the compiler is performed and the resulting decorated AST is passed to the prototypes *DefaultAnnotationTaskListener* which then utilizes the data-flow based analysis as described in Section 3. Again the detected errors are delegated to the compiler.

It has to be noted that the before-mentioned language model only represents a subset of Java. Language constructs which are embedded within method bodies such as assignments or method invocations are not included. Another API is required in order to access language constructs below the method level. Oracle's javac compiler provides a compiler-specific API called *compiler tree API* [Ora14]. This lower-level API provides a composite structure that represents the required *whole* AST created by javac.

5 Evaluation

The developed prototype has been used in several Java projects in order to demonstrate its functionality and to evaluate its performance.

All example applications are based on the Spring framework 3.11: The *Spring Pet Clinic*[3] is a sample application provided by the Spring framework. It is selected because it demonstrates the usage of all Spring features which are considered by the annotation processor such as annotation-based dependency injection, transaction management and caching. *Broadleaf Commerce*[4] is an open-source e-commerce framework based on Java and Spring, which consists of about 115.000

[3] See https://github.com/spring-projects/spring-petclinic

[4] See https://github.com/BroadleafCommerce/BroadleafCommerce

lines of code (LoC). It represents an actual real-world use case in which developers build modules of comparable size multiple times a day. Besides that, 12 examples[5] are considered, each of which includes an instance of one of the identified Spring-configuration error types. They do not represent practical applications but are used to ensure that the prototype is actually able to detect such errors.
Since the prototype has to be integrated into the third party projects (Spring Pet Clinic, Broadleaf Commerce), it is also possible to evaluate the integration efforts from the user perspective.

In order to assess the performance of the prototype, the build times of the projects including and excluding the prototype are compared to each other. Since all considered projects are based on Maven, the time measured by Maven itself is used. To ensure the comparability of the measured times, all builds are performed on the same machine and unchanged configuration. In addition, multiple builds are performed to minimize the possible influence of external factors of other processes performed by the operating systems.
Each project is build 41 times whereby the first build fulfills two functions: First, it is used by Maven to download and manage third party dependencies, which potentially falsifies the results. Second, the Java virtual machine requires some time for initialization when started, which also potentially falsifies the results. 40 additional builds are used to actually measure the build times. 20 of these builds are performed with the prototype and 20 are performed without the prototype.

The conducted tests reveal the following results. No Spring-related errors have been deteted within the Spring Pet Clinic and Broadleaf Commerce projects, which is not surprising, since they are sufficiently mature. In contrast, the errors placed on purpose in the example projects are detected.

The runtimes required to build the projects with and without the annotation processor differ by an acceptable amount (see Table 3). The time differences are less than one second for the small projects. Tests with an empty annotation processor, which performs no checks, show similar results. Hence, it is likely that a large part of the difference results from locating and initializing the annotation processor. The largest absolute difference observed at the Broadleaf Commerce project is $\sim$2 seconds, a 3% increase, for $115,000$ LoC. However, it has to be noted that the project consists of seven modular projects and the Java plugin is invoked individually for each of them. Hence, the annotation processor is seven times located and initialized.

Up to now, our prototype handles 12 of the identified error types and, as our experiments have shown, it is able to detect instances of these error types successfully. The implementation of the detection of remaining error types is still pending.

In order to improve the acceptance of our tool, the prototype is required to minimize the occurrences of false reports. The following can be stated w.r.t. the correctness and completeness of our tool. All errors above the method level are reported properly and there are no reports of errors which actually don't occur.

[5] See https://github.com/vvhof/DetectingSpringConfigurationErrorsExamples

		Avg. Build Times in ms		
Sample Project	**LoC**	**Disabled**	**Enabled**	**Diff.**
Error Type 1	29	2 568	2 971	16%
Error Type 2	156	2 675	2 932	10%
Error Type 3	36	2 544	2 741	8%
Error Type 4	37	2 535	2 825	11%
Error Type 5	37	2 524	2 870	14%
Error Type 6	69	2 524	2 915	15%
Error Type 7	56	2 558	2 756	8%
Error Type 8	53	2 501	2 629	5%
Error Type 9	39	2 463	2 725	11%
Error Type 10	39	2 469	2 668	8%
Error Type 11	54	2 502	2 764	11%
Error Type 12	54	2 516	2 748	9%
Spring PetClinic	1 390	9 912	11 448	15%
Broadleaf Commerce	115 902	55 108	56 970	3%

Table 3. Build Times with enabled and disabled prototype.

Below the method level, we are using a static analysis based on the control-flow graph in addition to an attributed grammar. Due to the unavoidable loss of precision in that analysis, it may happen that errors are reported, which cannot occur thanks to data dependencies which the reaching definitions analysis ignores. Fortunately, such errors rarely happen in practice, since it is bad programming style to let the correctness of the configuration depend on the control and data flow. One may even argue that such cases should be reported. Our current implementation does not yet support inter-method analysis. Thus, errors which can only be detected with such an analysis are currently not yet covered.

There are two limitations of our approach. As explained above, the limits of static code analysis are also the limits of our approach. Second, the pluggable annotation processing API restricts the usage of annotation processors to a Java compiler. The usage of the compiler tree API also binds the annotation processor to Oracle's specific Java compiler javac.

6 Conclusion and Future Work

Dependency injection is an elegant design pattern. However using it, configuration errors may occur which available Java compilers cannot detect. These erroneous configurations are hence only detected at runtime, which requires difficult debugging and causes nasty delays during the development of software. We have developed a compiler-plugin for the javac compiler which is able to find such configuration errors at compile time. Conceptually, the plugin is based on attributed grammars and the Java pluggable annotation processing API.

For the popular framework Spring and based on a literature review and expert interviews, we have first of all collected a set of 38 possible types of configuration errors. Then, we have classified these error types into four categories. The classification depends on two aspects. First, we check whether an error requires an analysis above or below the method level. Secondly, we check whether the error is depending on the Spring context or not. For each of these classes of errors, we have developed a scheme for an S- or L-attributed grammar and instantiated this scheme for every considered possible error. By combining the attributed grammars of each error type, we obtain one large L-attributed grammar. For errors which require an analysis of the control flow, a reaching definitions analysis based on the control-flow graph has been added.

In experiments based on a couple of small and two big applications, we have evaluated that the compiler plugin produces an acceptable runtime overhead. Moreover, it was able to find all configuration errors which we have inserted. Due to the imprecision of the reaching definitions analysis, false positives may happen in principle. In practice, this did not happen.

Our plugin is a valuable tool for Spring developers and used in practice by our project partner in industry. It has helped to speedup software development using Spring considerably.

Our current implementation handles 12 out of 38 identified types of errors. As future work, we would like to extend the plugin such that the remaining error types are also covered. For all but 5 error types, this will be straightforward and we just have to instantiate our general schemes again. The remaining 5 errors will require some inter-method analysis.

7 Acknowledgments

We thank our project partner viadee GmbH for the fruitful collaboration.

References

[ALSU07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools.* Addison-Wesley Publishing Company, USA, 2nd edition, 2007.

[APM+07] Nathaniel Ayewah, William Pugh, J David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 1–8. ACM, 2007.

[Bur03] Oliver Burn. Checkstyle, 2003.

[CK05] David R Cok and Joseph R Kiniry. Esc/java2: Uniting esc/java and jml. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 108–128. Springer, 2005.

[Dar06] Joseph D. Darcy, 2006.

[EH07] Torbjörn Ekman and Görel Hedin. The jastadd extensible java compiler. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 1–18, New York, NY, USA, 2007. ACM.

[GF07] David Greenfieldboyce and Jeffrey S Foster. Type qualifier inference for java. In *ACM SIGPLAN Notices*, volume 42, pages 321–336. ACM, 2007.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Joh15] Rod et al. Johnson. Spring framework reference documentation, 2015.

[Knu68] Donald E. Knuth. Semantics of context-free languages. In *Mathematical Systems Theory*, pages 127–145, 1968.

[LL12] Jochen Ludewig and Horst Lichter. *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken*. dpunkt. verlag, 2012.

[MME$^+$10] Shane Markstrum, Daniel Marino, Matthew Esquivel, Todd Millstein, Chris Andreae, and James Noble. Javacop: Declarative pluggable types for java. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(2):4, 2010.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[Ora14] Oracle Corporation. Openjdk compiler tree api specification, 2014.

[Ora15] Oracle Corporation. ”package java.lang.reflect”, 2015.

[PAC$^+$08] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa, Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for java. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA ’08, pages 201–212, New York, NY, USA, 2008. ACM.

[Piv15] Pivotal Sofware, Inc. Spring framework, 2015.

[PMD15] PMD. Pmd, 2015.

[Pra09] Dhanji R. Prasanna. *Dependency Injection*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2009.

[SDE12] Eric Spishak, Werner Dietl, and Michael D. Ernst. A type system for regular expressions. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, FTfJP ’12, pages 20–26, New York, NY, USA, 2012. ACM.

[SK95] Kenneth Slonneger and Barry Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.

[Son15] SonarSource S.A. Sonarqube, 2015.

[VWKBJ06] Eric Van Wyk, Lijesh Krishnan, Derek Bodin, and Eric Johnson. Adding domain-specific and general purpose language features to java with the java language extender. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA ’06, pages 728–729, New York, NY, USA, 2006. ACM.

[Wal14] Craig Walls. *Spring in Action*. Manning Publications Co., Greenwich, CT, USA, 4th edition, 2014.

Kleenex: Compiling Nondeterministic Transducers to Deterministic Streaming Transducers

Bjørn Bugge Grathwohl Fritz Henglein
Ulrik Terp Rasmussen Kristoffer Aalund Søholm
Sebastian Paaske Tørholm
DIKU, University of Copenhagen
{bugge,henglein,dolle,soeholm,sebbe}@diku.dk

September 18th, 2015

Abstract

We present and illustrate Kleenex, a language for expressing general nondeterministic finite transducers, and its novel compilation to streaming string transducers with essentially optimal streaming behavior, worst-case linear-time performance and sustained high throughput. In use cases it achieves consistently high throughput rates around the 1 Gbps range on stock hardware, performing well, especially in complex use cases, in comparison to both specialized and related tools such as `AWK`, `sed`, `grep`, RE2, Ragel and regular-expression libraries.

1 Introduction

Imagine you want to implement syntax highlighting. This can be thought of as parsing the input into its tokens and processing each token according to its class. For illustration, assume we have one keyword, `for`, and alphabetic identifiers as the only tokens. The lexical structure of the input is essentially described by the *regular expression (RE)* $((\texttt{for}|[a-z]^*)\ )^*$, where whitespace is, for simplicity, represented by the single blank between the two closing parentheses. This scenario highlights the following:

Ambiguity by design. The RE is ambiguous. The intended semantics is that the left alternative has higher priority than the right. This is *greedy* disambiguation: Choose the left alternative if possible, treating e^* as its *unfolding* $ee^*|1$. Accordingly, in our example `for` matches the left alternative, not the right.

Regular expression parsing. Note that the RE has star height 2; in particular, we need to *parse* the input under multiple Kleene stars. For our RE the parse of a string is a list of segments (corresponding to the outer Kleene star), with each segment represented by a pair, a token and whitespace (corresponding to concatenation), where each token is tagged (corresponding to the alternation) to indicate that it is either the keyword `for` or an identifier; an identifier, in turn, consists of a list (corresponding to the inner Kleene star) of characters.

Output actions. We need to output something, the highlighted tokens, not just *accept* or *reject* a string as is done by finite *automata*. Note that output actions are not specified in our RE.

We would like to do the highlighting in a *streaming* fashion, using as little internal storage as possible and performing output actions as early as they are determined by the input prefix read so far, at a high sustained input processing rate, in particular in worst-case linear-time in the length of the input stream with a low factor depending linearly on the size of the RE. We would like to accomplish this automatically for *arbitrary* REs (or similar input format specification) and output actions, with speeds that in practice adapt to how much output actually needs to be produced; in particular, performance should gracefully approach pure acceptance testing as more and more output actions are removed. How?

It turns out that the set of parses are exactly the elements of the RE read as a *type* [**?**, **?**]: Kleene-star is the (finite) list type constructor, concatenation the Cartesian product, alternation the sum type and an individual character the singleton type containing that character. A *Thompson automaton* [**?**] represents an RE in a strong sense: the complete paths—paths from initial to final state—are in one-to-one correspondence with the parses [**?**]. If a string has 4 parses (e.g. "for for "), then there are exactly 4 complete paths accepting it. Let us look at bit closer at a Thompson automaton: It is nondeterministic, with ϵ-transitions, easily constructed, having $O(m)$ states and transitions from an RE of size m. It has exactly one initial and one accepting node. Every state is either *nondeterministic*: it has two outgoing ϵ-transitions ("left" or "right"); or it is *deterministic*: it has exactly one outgoing transition labeled by ϵ or an input symbol, or it is the final state, which has no outgoing transition. Every complete path is determined by a sequence of bits used as an oracle [**?**]. Starting with the initial state, follow all outgoing transitions from deterministic states; upon arriving at a nondeterministic state query the oracle to determine whether to go left or right, until the final state is reached. The bit sequence of query responses yields a prefix-free binary code for the string accepted on the designated path. This *bit-code* can also be computed directly from the RE underlying the Thompson NFA [**?**, **?**]. Since a bit-code represents a particular parse, a string can have multiple bit-codes if and only if the RE (and thus Thompson automaton) is ambiguous: The *greedy* parse of a string, which we are interested in, corresponds to the *lexicographically least* amongst its bit-codes [**?**].

The *greedy RE parsing problem* is producing this lexicographically least bit-code for a string matching a given RE. This can be done by an *optimally streaming* algorithm, running in time linear in the size of the input string for fixed RE [**?**]: The bits in the output are produced as soon as they are uniquely determined by the input prefix read so far, assuming the input string will eventually be accepted. The algorithm maintains an ordered *path tree* from the initial state to all the automata states reachable by the input prefix read so far. A branching node represents both sides of an alternation that are both still viable. The (possibly empty) path segment from the initial state to the first branching node is what can be output based on the input prefix processed so far, without knowing which of the presently reached states will eventually accept the rest of the input. This works for all REs and all inputs; e.g., it automatically results in constant memory space consumption for REs that are deterministic modulo finite look-ahead, e.g. one-unambiguous REs [**?**].

Let us step back a bit. It is possible to aggressively ("earliest possible") and efficiently stream out the bit-code of the greedy parse of an input string under a given RE as the input is streaming in: worst-case linear time in the input string size, no backtracking and each input symbol can be processed in time $O(m)$, linear in the size of the RE and of its Thompson NFA. (Here it is critical that Thompson NFAs have ϵ-transitions since equivalent ϵ-free automata require $\Omega(m \log m)$ transitions [**?**] and standard ϵ-free NFA-constructions [**?**, **?**, **?**] even $\Omega(m^2)$.)

Coming back to our syntax highlight problem we can use this algorithm to parse the input, build the parse tree from the bit-code and recursively descend it to perform the syntax highlighting. We might (correctly) suspect that the highlighting can be done by piping the bit-code into a separate highlighter process, eliding the materialization of the bit-code.[1] In this paper we show we can do better yet: The algorithm can be generalized to simulating arbitrary nondeterministic finite-state transducers, NFAs with output actions. Furthermore, we can compile their nondeterminism away by producing theoretically and practically very efficient streaming string transducers [**?**, **?**, **?**].

1.1 Contributions

This paper makes the following novel contributions:

- An aggressively *streaming* algorithm for *nondeterministic finite state transducers (NFST)* for ordered output alpabets, which emits the lexicographically least output sequence generated by all accepting paths

[1] All Kleenex code in this paper was highlighted with a Kleenex program emitting LaTeX-commands.

of an input string. It runs in $O(mn)$ time, for automata of size m and inputs of size n.

- An effective determinization of NFSTs into a subclass of *streaming string transducers (SST)* [?], finite state machines with string registers that are updated linearly when entering the state upon reading an input symbol. The number of registers required adapts to the number of output actions in the NFST: The fewer output actions the fewer registers. In particular, without special-casing, no registers are generated—yielding a deterministic finite automata (DFA).
- An expressive declarative language, *Kleenex*, for specifying NFSTs with full support for and clear semantics of unrestricted nondeterminism by greedy disambiguation. A basic Kleenex program is a context-free grammar with embedded semantic output actions, but syntactically restricted to ensure that the input is regular.[2] Basic Kleenex programs can be functionally composed into pipelines. The central technical aspect of Kleenex is its semantic support for unbridled (regular) nondeterminism and its effective determinization and compilation to SSTs, thus both highlighting and complementing their significance.
- An implementation, including some empirically evaluated optimizations, of Kleenex that generates SSTs and sequential machines rendered as standard single-threaded C-code, which is eventually compiled to X86 machine code. The optimizations, which are neither conclusive nor final, illustrate the design robustness obtained by the underlying theories of ordered NFST's and SST's.
- Use cases that illustrate the expressive power of Kleenex, and a performance comparison with related tools, including Ragel [?], RE2 [?] and specialized string processing tools. These document Kleenex's consistently high performance (typically around 1 Gbps, single core, on stock hardware) even when compared to expressively more specialized tools with special-cased algorithms and tools with no or limited support for nondeterminism.

2 Transducers

The semantics of Kleenex will be given by translation to non-deterministic *finite state transducers*, which are finite automata extended with output in a free monoid. In this section, we will recall the standard definition (see e.g. Berstel [?]). Since Kleenex is deterministic, we also need to define a disambiguated semantics which allows us to interpret any non-deterministic transducer as a partial function, even when it may have more than one

[2] This facilitates avoiding the $\Omega(M(n))$ lower bound for context-free grammar parsing, where $M(n)$ is the complexity of multiplying $n \times n$ matrices.

possible output for a given input string.

In the following, an *alphabet* is understood to be a finite subset $\{0, 1, ..., n-1\} \subseteq \mathbb{N}$ of consecutive natural numbers with their usual ordering. We fix two alphabets Σ and Γ called the *input* and *output* alphabets, respectively.

Definition 1 (Finite State Transducer)**.** A *finite state transducer* (FST) over Σ, Γ is a structure $\mathcal{T} = (\Sigma, \Gamma, Q, q^-, q^f, E)$ where

- Q is a finite set of *states*;
- $q^-, q^f \in Q$ are the *initial* and *final* states, respectively;
- $E : Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \times Q$ is the *transition relation.*

We write $q \xrightarrow{x|y} q'$ whenever $(q, x, y, q') \in E$. The *support* of $q \in Q$ is defined as $\mathrm{supp}(q) = \{x \in \Sigma \cup \{\epsilon\} \mid \exists q', v.\, q \xrightarrow{x|v} q'\}$.

A *path* in $\mathcal{T}$ is a sequence of transitions

$$q_0 \xrightarrow{x_1|y_1} q_1 \xrightarrow{x_2|y_2} ... \xrightarrow{x_n|y_n} q_n$$

It has *input label* $u = x_1x_2...x_n$ and *output label* $v = y_1y_2...y_n$ (ϵ denotes the empty string). We write $q_0 \xrightarrow{u|v} q_n$ if a path from q_0 to q_n with input label u and output label v exists.

$\mathcal{T}$ is *normalized* if for every state $q \in Q_{\mathcal{T}}$, either $\mathrm{supp}(q) = \{\epsilon\}$ or $\mathrm{supp}(q) \subseteq \Sigma$; and furthermore $\mathrm{supp}(q^f) \subseteq \Sigma$. We write $q\downarrow$ for q such that $\mathrm{supp}(q) \subseteq \Sigma$. The formulation of our simulation algorithm in Section 4 becomes simpler when restricting our attention to normalized transducers, since we can take advantage of the following separation property:

Proposition 1. *If $\mathcal{T}$ is normalized, then $p \xrightarrow{uv|z} r\downarrow$ if and only if there exists a q such that $z = xy$ and $p \xrightarrow{u|x} q\downarrow \xrightarrow{v|y} r\downarrow$.*

Definition 2 (Relational Semantics)**.** An FST $\mathcal{T}$ denotes a relation $[\![\mathcal{T}]\!] \subseteq \Sigma^* \times \Gamma^*$ with $(u, v) \in [\![\mathcal{T}]\!]$ iff $q^- \xrightarrow{u|v} q^f$.

The relations definable as FSTs are the *rational relations* [**?**]. In the special case where for any $u \in \Sigma^*$ there is at most one $v \in \Gamma^*$ such that $(u, v) \in [\![\mathcal{T}]\!]$, the transducer computes a partial function. Any FST can be translated to an equivalent normalized FST.

In the following we give a refined semantics which allows us to interpret any FST as denoting a partial function, using the assumed ordering on alphabets to disambiguate between outputs. Our semantics requires restricting paths to be nonproblematic [**?**]: If a path contains a non-empty loop $q' \xrightarrow{\epsilon|v'} q'$ with empty input label, then the path is said to be *problematic*, otherwise it is *nonproblematic*. If there is a nonproblematic path from q to q' with labels u, v, then we write a subscript on the arrow: $q \xrightarrow{u|v}_{\mathsf{np}} q'$.

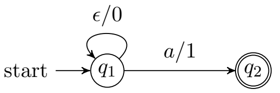
The output words (elements of Γ^*) are lexicographically ordered: $w_1 \leq w_2$ if either w_1 is a prefix of w_2, or there exist words w', w_1', w_2' and symbols $b_1, b_2 \in \Gamma$ such that $w_1 = w' b_1 w_1'$, $w_2 = w' b_2 w_2'$ and $b_1 < b_2$. We use the ordering on output words to choose a single path from a non-empty set of paths:

Definition 3 (Functional Semantics)**.** Any transducer $\mathcal{T}$ denotes a partial function $[\![\mathcal{T}]\!]_{\leq} : \Sigma^* \to \Gamma^* \cup \{\emptyset\}$ where

$$[\![\mathcal{T}]\!]_{\leq}(u) = \min\{v \mid q^- \xrightarrow{u|v}_{\mathsf{np}} q^f\}.$$

Note that a partial function $A \to B \cup \{\emptyset\}$ is considered here to be a map $A \to \mathbb{2}^B$ where the cardinality of every subset in its range is at most two, and we tacitly identify elements $a \in A$ with their singleton sets $\{a\}$.[3]

Why the restriction to nonproblematic paths? Consider the following transducer $\mathcal{T}$:



Then $\min\{v \mid q_1 \xrightarrow{a|v} q_2\} = \emptyset$, as evidenced by the following infinitely descending chain of outputs: $1 \geq 01 \geq 001 \geq 0001 \geq ....$ Operationally, such a chain corresponds to a non-terminating backtracking search. On the other hand, the number of nonproblematic paths with a given input label is always finite, ensuring well-foundedness of the lexicographic order. Every problematic path has a corresponding nonproblematic path with the same input label; consequently, $\mathrm{dom}([\![\mathcal{T}]\!]_{\leq}) = \mathrm{dom}([\![\mathcal{T}]\!])$.

3 Kleenex

The core syntax of Kleenex is essentially that of right regular grammars extended with *output actions* and choice operators. Semantically, a Kleenex program denotes a function which transforms an input string from a regular language into a sequence of action symbols, with the caveat that if the input grammar is ambiguous, then the production rules are chosen according to a greedy leftmost disambiguation strategy.

We will first present the abstract syntax of core Kleenex, which is given a semantics in terms of the transducers introduced in Section 2.

Definition 4 (Kleenex syntax)**.** Let Σ and Γ be two alphabets. A Kleenex program is a non-empty list $p = d_0 d_1 \ldots d_n$ of *definitions* d_i, each of the form $N_i \mathtt{:=}\, t_i$, where t_i is a *term* generated by the grammar:

$$t ::= \mathtt{1} \mid N \mid a\ t \mid \texttt{"}b\texttt{"}\ t \mid t_0 \,\mathtt{|}\, t_1$$

[3] In other words, we adjoin an element to model partial functions as total functions to *pointed sets*.

In the above, N is assumed to range over some set of non-terminal identifiers $\{N_1, ..., N_n\}$, $a \in \Sigma$ over *input symbols* and $b \in \Gamma$ over *output actions*.

We restrict the valid Kleenex programs to those where there is at most one definition for each non-terminal identifier.

Let p be a Kleenex program over non-terminals $\{N_1, \dots, N_n\}$. We define a set of states Q_p and two transition relations $E_p^{\mathsf{A}}, E_p^{\mathsf{C}}$ as the smallest sets closed under the following rules:

$$\frac{}{N_1 \in Q_p} \qquad \frac{N_i \in Q_p}{t_i \in Q_p \quad (N_i, \epsilon, \epsilon, t_i) \in E_p^{\mathsf{A}} \cap E_p^{\mathsf{C}}} (N_i \,\mathtt{:=}\, t_i)$$

$$\frac{a\ t \in Q_p}{t \in Q_p \quad (a\ t, a, \epsilon, t) \in E_p^{\mathsf{C}} \quad (a\ t, \epsilon, \epsilon, t) \in E_p^{\mathsf{A}}}$$

$$\frac{\mathtt{"}b\mathtt{"}t \in Q_p}{t \in Q_p \quad (\mathtt{"}b\mathtt{"}\ t, \epsilon, \epsilon, t) \in E_p^{\mathsf{C}} \quad (\mathtt{"}b\mathtt{"}\ t, \epsilon, b, t) \in E_p^{\mathsf{A}}}$$

$$\frac{t_0 \,\mathtt{|}\, t_1 \in Q_p}{\begin{array}{ll} \{t_0, t_1\} \subseteq Q_p & (t_0 \,\mathtt{|}\, t_1, \epsilon, 0, t_0), (t_0 \,\mathtt{|}\, t_1, \epsilon, 1, t_1) \in E_p^{\mathsf{C}} \\ & (t_0 \,\mathtt{|}\, t_1, 0, \epsilon, t_0), (t_0 \,\mathtt{|}\, t_1, 1, \epsilon, t_1) \in E_p^{\mathsf{A}} \end{array}}$$

The sets are easily seen to be finite. They define two transducers, an *oracle* $\mathcal{T}_p^{\mathsf{C}} = (\Sigma, \mathbb{2}, Q_p, N_1, \mathtt{1}, E_p^{\mathsf{C}})$ and an *action machine* $\mathcal{T}_p^{\mathsf{A}} = (\mathbb{2}, \Gamma, Q_p, N_1, \mathtt{1}, E_p^{\mathsf{A}})$, where $\mathcal{T}_p^{\mathsf{A}}$ is easily seen to be deterministic, and $\mathcal{T}_p^{\mathsf{C}}$ is non-deterministic and possibly ambiguous. The oracle intuitively translates an input string to a set of codes for the possible paths through p which reads the given string. The action machine translates a code to a sequence of actions.

Disambiguating according to the greedy leftmost strategy corresponds to choosing the lexicographically least code, and we can thus define the semantics as follows:

Definition 5 (Kleenex semantics)**.** Let p be a Kleenex program and let $\mathcal{T}_p^{\mathsf{C}}$ and $\mathcal{T}_p^{\mathsf{A}}$ be defined as above. The program p denotes a partial function $\llbracket p \rrbracket : \Sigma^* \to \Gamma^* \cup \{\emptyset\}$ given by

$$\llbracket p \rrbracket = \llbracket \mathcal{T}_p^{\mathsf{A}} \rrbracket \circ \llbracket \mathcal{T}_p^{\mathsf{C}} \rrbracket_{\leq}$$

3.1 Syntactic sugar

The full syntax of our language is obtained by extending the syntax of core Kleenex with the following term-level constructors:

$$\begin{aligned} t ::= {} & \dots \mid \mathtt{"}v\mathtt{"} \mid \mathtt{/}e\mathtt{/} \mid \mathtt{\sim}t \mid t_0 \cdot t_1 \mid t\mathtt{*} \mid t\mathtt{+} \mid t\mathtt{?} \\ & \mid t\mathtt{\{}n\mathtt{\}} \mid t\mathtt{\{}n\mathtt{,\}} \mid t\mathtt{\{,}m\mathtt{\}} \mid t\mathtt{\{}n\mathtt{,}m\mathtt{\}} \end{aligned}$$

```
main     := odd  ~/a/
          | even ~/a/
odd      := ~/aa/ "bb" odd
          | "c"
even     := ~/a/ "c" even
          | "b"
```

$$\begin{aligned} N_{\text{main}} &\mathrel{:=} N_{\text{odd}} \mathbin{|} N_{\text{even}} \\ N_{\text{odd}} &\mathrel{:=} a\ a\ \texttt{"}b\texttt{"}\ \texttt{"}b\texttt{"}\ N_{\text{odd}} \\ &\qquad \mathbin{|}\texttt{"}\ c\ \texttt{"}\ a\ \mathtt{1} \\ N_{\text{even}} &\mathrel{:=} a\ \texttt{"}c\texttt{"}\ N_{\text{even}} \mathbin{|} \texttt{"}b\texttt{"}\ a\ \mathtt{1} \end{aligned}$$

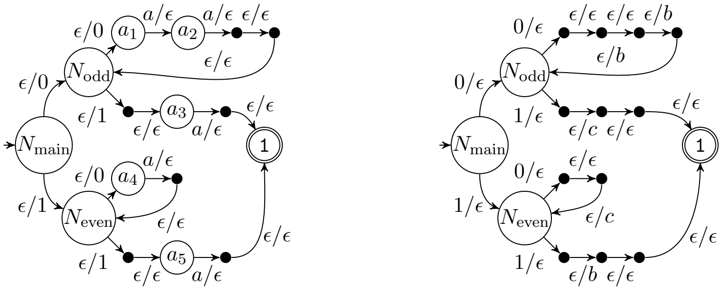


Figure 1: In the top left is a Kleenex program in the surface syntax and on the right is the desugared version. Below, the oracle transducer and action machine is shown, from left to right. The transduction realized by the program maps a^{2n+1} to $b^{2n}c$, and a^{2n+2} to $c^{2n}b$, respectively.

where $v \in \Gamma^*$, $n, m \in \mathbb{N}$, and e is a *regular expression.* The term "v" is just shorthand for a sequence of action symbols. The regular expressions are special versions of Kleenex terms that do not contain identifiers. They always desugar to terms that output the matched input string: The sugared term /e/ adds a default action "$\alpha(a)$" after every input symbol a in e using a given default action map $\alpha : \Sigma \to \Gamma$. For example, the regular expression `/a*|b{n,m}|c?/` becomes the term $(a\texttt{"}a\texttt{"})\texttt{*}\mathbin{|}(b\texttt{"}b\texttt{"})\{n\texttt{,}m\}\mathbin{|}(c\texttt{"}c\texttt{"})\texttt{?}$. A *suppressed* subterm is written ~t, and it desugars into t with all action symbols removed. Composition $t_0 \cdot t_1$ allows general sequential composition instead of the strict cons-like form of the core syntax. The operators $\cdot\texttt{*}$, $\cdot\texttt{+}$ and $\cdot\texttt{?}$ desugar to their usual meaning as regular operators, as do operators $\cdot\{n\}$, $\cdot\{n\texttt{,}\}$, $\cdot\{\texttt{,}m\}$, and $\cdot\{n\texttt{,}m\}$.

By convention, the nonterminal named `main` is the entry point to a Kleenex program.

The desugaring can be described more precisely by a desugaring operator $\mathcal{D}[\![\cdot,\cdot]\!]$. The result of desugaring a program $p = d_1 \dots d_n$ with initial term $N_1 \mathrel{:=} t_1$ is a program with initial term $N_1' \mathrel{:=} \mathcal{D}[\![t_1, \mathtt{1}]\!]$ which furthermore is a solution to the following set of equations:

$$\begin{aligned}
\mathcal{D}\llbracket \texttt{1}, k\rrbracket &= \mathcal{D}\llbracket \texttt{\~{}1}, k\rrbracket = k \\
\mathcal{D}\llbracket \texttt{"}b_1 \ldots b_n\texttt{"}, k\rrbracket &= \texttt{"}b_1\texttt{"} \ldots \texttt{"}b_n\texttt{"}\ k \\
\mathcal{D}\llbracket \texttt{\~{}}(\texttt{"}b\texttt{"}\ t), k\rrbracket &= \mathcal{D}\llbracket \texttt{\~{}}t, k\rrbracket \\
\mathcal{D}\llbracket a\ t, k\rrbracket &= a\ \mathcal{D}\llbracket t, k\rrbracket \\
\mathcal{D}\llbracket \texttt{\~{}}(a\ t), k\rrbracket &= a\ \mathcal{D}\llbracket \texttt{\~{}}t, k\rrbracket \\
\mathcal{D}\llbracket \texttt{\~{}}(t_0 \,\texttt{|}\, t_1), k\rrbracket &= \mathcal{D}\llbracket \texttt{\~{}}t_0, k\rrbracket \,\texttt{|}\, \mathcal{D}\llbracket \texttt{\~{}}t_1, k\rrbracket \\
\mathcal{D}\llbracket N, k\rrbracket &= N_{\mathcal{D}\llbracket t,k\rrbracket} && \text{(where } N \texttt{:=}\ t\text{)} \\
\mathcal{D}\llbracket \texttt{\~{}}N, k\rrbracket &= N_{\mathcal{D}\llbracket \texttt{\~{}}t,k\rrbracket} && \text{(where } N \texttt{:=}\ t\text{)} \\
\mathcal{D}\llbracket \texttt{/}e\texttt{/}, k\rrbracket &= \mathcal{D}\llbracket t_e, k\rrbracket \\
\mathcal{D}\llbracket t_0 \cdot t_1, k\rrbracket &= \mathcal{D}\llbracket t_0, \mathcal{D}\llbracket t_1, k\rrbracket\rrbracket \\
\mathcal{D}\llbracket t_0 \,\texttt{|}\, t_1, k\rrbracket &= \mathcal{D}\llbracket t_0, k\rrbracket \,\texttt{|}\, \mathcal{D}\llbracket t_1, k\rrbracket \\
\mathcal{D}\llbracket t\texttt{*}, k\rrbracket &= \mathcal{D}\llbracket t, \mathcal{D}\llbracket t\texttt{*}, k\rrbracket\rrbracket \,\texttt{|}\, k \\
\mathcal{D}\llbracket t\texttt{+}, k\rrbracket &= \mathcal{D}\llbracket t, \mathcal{D}\llbracket t\texttt{*}, k\rrbracket\rrbracket \\
\mathcal{D}\llbracket t\texttt{?}, k\rrbracket &= \mathcal{D}\llbracket t, k\rrbracket \,\texttt{|}\, k
\end{aligned}$$

In the above, a non-terminal name N_t on the right-hand side of an equation implies the presence of a definition $N_t \texttt{:=}\ t$, and the term t_e corresponds to the regular expression e as described above. The range patterns are just expanded and then further desugared.

The system does not always have a well-defined solution: The generalized composition operator of sugared Kleenex allows one to write non-regular grammars, for example:

$$A \texttt{:=}\ (aA) \cdot b \,\texttt{|}\, \texttt{1}.$$

A program that does not have a well-defined desugaring is not considered to be well-formed, and we will not attempt to give it a semantics.

3.2 Custom register updates

We extend the syntax of Kleenex further with *register actions*:

$$\begin{aligned}
t ::= \ldots\ &\mid R\ \texttt{@}\ t \mid \texttt{!}R \\
&\mid \texttt{[}\ R\ \texttt{<-}\ (R \mid \texttt{"}v\texttt{"})^\star\ \texttt{]} \\
&\mid \texttt{[}\ R\ \texttt{+=}\ (R \mid \texttt{"}v\texttt{"})^\star\ \texttt{]}
\end{aligned}$$

where R is a lower-case register name. Intuitively, these constructs allow one to store actions and perform them later. Writing $R\ \texttt{@}\ t$ redirects all actions that would have resulted from running t into the register R, which can be performed later by writing $\texttt{!}R$. The register R can be either set to a sequence of actions $(R \mid \texttt{"}v\texttt{"})^\star$ or appended with them, using the `<-` and `+=` construct, respectively.

At first glance it seems like adding custom register updates to Kleenex significantly alters the language and moves beyond the semantics discussed so far. However, the only requirement on the output actions is that they form a monoid, so this is not the case. We simply add actions like "set register R to v" as output symbols along with the output symbols $\texttt{!}v$. The output redirection caused by the $\cdot \mathbin{\texttt{@}} \cdot$ operator can be understood as a push operation: when $R \mathbin{\texttt{@}} t$ is written it means that in the scope of t the topmost register is R. If there are other redirection symbols in t, these will come in and out of scope as they are pushed and popped to the stack.

As an example, the following program swaps two input lines by storing them in registers `a` and `b` and outputting them in reverse order:

```
main := a@line b@line !b !a
line := /[^\n]*\n/
```

4 Simulation and determinization

In this section, we specify an algorithm for simulating FSTs under the functional semantics, allowing us to efficiently evaluate the oracle transducer defined in Section 3. We also show how the simulation algorithm can be implemented by finite deterministic *streaming string transducers* [**?**] whose states are identified by equivalence classes of simulation states. The latter construction gives a deterministic machine model for Kleenex programs which can be compiled to efficient code for executing on hardware.

We note that non-deterministic transducers are strictly more powerful than their deterministic counterparts, and can thus not always be determinized in general. Determinization procedures exist [**?**, **?**] which result in a deterministic transducer with an infinite state set in the general case, and a finite state set if and only if the underlying transduction is *subsequential* [**?**, **?**]. The oracle transducers of Kleenex programs are not subsequential in general. Our simulation algorithm is also different from the existing methods for determinizing transducers by also taking disambiguation into account.

In the following we fix a transducer $\mathcal{T} = (\Sigma, \Delta, Q, q^-, q^f, E)$. We will assume that $\mathcal{T}$ is normalized, and that it furthermore satisfies the following property:

Definition 6 (Prefix-free transducer)**.** $\mathcal{T}$ is said to be *prefix-free* if for all $p, q, q' \in Q_\mathcal{T}$ where $\operatorname{supp}(q), \operatorname{supp}(q') \subseteq \Sigma$ we have that if $p \xrightarrow{x|y} q$ and $p \xrightarrow{x|y'} q'$ then $y \not\prec y'$.

It is easy to verify that the oracle transducers constructed Section 3 are both normalized and prefix-free. Note that they will always have $\Delta = \mathbb{2}$, but our construction generalizes to oracle alphabets of all sizes.

4.1 Generalized state set simulation

Let D be a finite and totally ordered set, and write $S(D, Q)$ for the set of partial functions $Q \to D^* \cup \{\emptyset\}$. Elements $A \in S(D, Q)$ can be seen as generalized subsets of Q where every member q is labeled by some element $A(q) \in D^*$, and every non-member has $A(q) = \emptyset$.

We extend word concatenation in D^* to the set $D^* \cup \{\emptyset\}$ by setting $x\emptyset = \emptyset = \emptyset x$. For $u, v \in D^* \cup \{\emptyset\}$, write $u \preceq v$ if u is a *prefix* of v, i.e. there is a unique w such that $v = uw$. Write $u \prec v$ if w has length at least one. Let $u \wedge v$ refer to the longest p such that $u = pu'$ and $v = pv'$ for some u', v'. Note that in view of this definition, $\emptyset$ becomes a neutral element with $u \wedge \emptyset = u = \emptyset \wedge u$.

We define a right action $\cdot : S(D, Q) \times \Sigma^* \to S(D \cup \Delta, Q)$ on the generalized state sets as follows:

Definition 7 (Right action)**.** Let $A \in S(D, Q)$ and $u \in \Sigma^*$. We define

$$(A \cdot u)(q) = \min\{A(p)v \mid p \xrightarrow{u|v}_{\mathsf{np}} q \downarrow\}.$$

When $D = \Delta$ the right action can be seen as a map $S(\Delta, Q) \times \Sigma^* \to S(\Delta, Q)$. It is easily seen that the right action is related to the functional semantics in the following way:

Proposition 2. *Let $A(q) = \epsilon$ if $q = q^-$ and $A(q) = \emptyset$ otherwise. Then $(A \cdot u)(q^f) = [\![\mathcal{T}]\!]_{\leq}(u)$.*

A generalized subset $A \in S(D, Q)$ is said to be *prefix-free* if $A(p) \not\prec A(q)$ for all $p, q \in Q$. When $\mathcal{T}$ is normalized and prefix-free, the right action preserves prefix-freeness of generalized subsets and commutes with word concatenation:

Proposition 3. *If $\mathcal{T}$ is normalized and prefix-free and A is prefix-free, then for all $u, v \in \Sigma^*$,*

1. *$A \cdot u$ is prefix-free; and*
2. *$(A \cdot u) \cdot v = A \cdot uv$.*

Proof. The first property follows directly by A and $\mathcal{T}$ being prefix-free. For the second, we have for $r \in Q$,

$$\begin{aligned}
&((A \cdot u) \cdot v)(r) \\
=\ &\min\{(A \cdot u)(q)y \mid q \xrightarrow{v|y}_{\mathsf{np}} r \downarrow\} \\
=\ &\min\{\min\{A(p)x \mid p \xrightarrow{u|x}_{\mathsf{np}} q \downarrow\}y \mid q \xrightarrow{v|y}_{\mathsf{np}} r \downarrow\} \\
=\ &\min\{\min\{A(p)xy \mid p \xrightarrow{u|x}_{\mathsf{np}} q \downarrow\} \mid q \xrightarrow{v|y}_{\mathsf{np}} r \downarrow\} \\
=\ &\min\{A(p)xy \mid p \xrightarrow{u|x}_{\mathsf{np}} q \downarrow \xrightarrow{v|y}_{\mathsf{np}} r \downarrow\} \\
=\ &\min\{A(p)z \mid p \xrightarrow{uv|z}_{\mathsf{np}} r \downarrow\} = (A \cdot uv)(r)
\end{aligned}$$

The third equality is a consequence of the fact that A and $\mathcal{T}$ are prefix-free, together with the following easily proved fact about lexicographic ordering: $(\min X)y = \min\{xy \mid x \in X)$ whenever X is a set of pairwise prefix-free words. The fourth equality is just associativity of minimum, and the last equality follows by the fact that $\mathcal{T}$ is normalized and Proposition 1. □

For $x \in D^*$ and $A \in S(D, Q)$, define $xA \in S(D, Q)$ by $(xA)(q) = x(A(q))$. We say that x is a prefix of A if $A = xA'$ for some A', which is equivalent to x being a prefix of every $A(q)$. The right action commutes with the prefix operation:

Proposition 4. *Let $x \in D^*$, then $(xA) \cdot u = x(A \cdot u)$ for all $u \in \Sigma^*$.*

Proof. Follows by the fact that lexicographic ordering satisfies $\min\{xy \mid y \in Y\} = x \min Y$. □

Streaming simulation algorithm A streaming simulation algorithm on $\mathcal{T}$ processes an input from left to right and may write zero or more symbols to the output in each step.

Algorithm 1 (Streaming FST Simulation)**.** Let $\mathcal{T}$ be a normalized and prefix-free transducer, and let the input $u = a_1 a_2 ... a_n$ be given. Let $A_0 \in S(\Delta, Q)$ be defined as in Proposition 2. Reading symbol a_i, compute $B_i = A_i \cdot a_{i+1}$. Append $p_i = \bigwedge_{q \in Q} B_i(q)$ to the output stream and set $A_{i+1} = B_i'$, where the equality $B_i = p_i B_i'$ defines B_i'. When there are no more input symbols left, append $(A_n \cdot \epsilon)(q^f)$ to the output and return, or fail if $(A_n \cdot \epsilon)(q^f) = \emptyset$.

By Proposition 2, Proposition 3 and Proposition 4, the algorithm computes $[\![\mathcal{T}]\!]_{\leq}(u)$.

4.2 A deterministic computation model

We wish to translate Kleenex programs to completely deterministic programs without a simulation overhead.

Single-valued transducers in general, however, can only be determinized if the underlying function is *subsequential* [**?**, **?**], a property which is not satisfied in general for the oracle transducers constructed from Kleenex programs.

We turn instead to *streaming string transducers* [**?**] (SST), a deterministic model of computation which generalizes subsequential transducers by allowing copy-free updates to a finite set of word registers. It turns out that every transducer that can be simulated by our generalized state set algorithm can be expressed as an SST.

Definition 8 (Streaming String Transducer [**?**])**.** A deterministic *streaming string transducer* (SST) over alphabets Σ, Δ is a structure $\mathcal{S} = (X, Q, q^-, F, \delta^1, \delta^2)$ where

- X is a finite set of *register variables*;
- Q is is a finite set of *states*;
- F is a partial function $Q \to (\Delta \cup X)^* \cup \{\emptyset\}$ mapping each *final state* $q \in \mathrm{dom}(F)$ to a word $F(q) \in (\Delta \cup X)^*$ such that each $x \in X$ occurs at most once in $F(q)$;
- δ^1 is a transition function $Q \times \Sigma \to Q$;
- δ^2 is a *register update* function $Q \times \Sigma \times X \to (\Delta \cup X)^*$ such that for each $q \in Q$, $a \in \Sigma$ and $x \in X$, there is at most one $y \in X$ such that x occurs in $\delta^2(q, a, y)$.

The semantics are defined as follows. A *configuration* of an SST $\mathcal{S}$ is a pair (q, ρ) where $q \in Q_\mathcal{S}$ is a state, and $\rho : X_\mathcal{S} \to \Delta^*$ is a *valuation*. A valuation extends as a monoid homomorphism to a map $\widehat{\rho} : (X_\mathcal{S} \cup \Delta)^* \to \Delta^*$ by setting $\rho(x) = x$ for $x \in \Delta$. The initial configuration is (q^-, ρ^-) where $\rho^-(x) = \epsilon$ for all $x \in X_\mathcal{S}$.

A configuration steps to a new configuration given an input symbol: $\delta_\mathcal{S}((q, \rho), a) = (\delta^1_\mathcal{S}(q, a), \rho')$, where $\rho'(x) = \widehat{\rho}(\delta^2_\mathcal{S}(q, a, x))$. The transition function extends to a transition function on words $\delta^*_\mathcal{S}$ by $\delta^*_\mathcal{S}((q, \rho), \epsilon) = (q, \rho)$ and $\delta^*_\mathcal{S}((q, \rho), au) = \delta^*_\mathcal{S}(\delta_\mathcal{S}((q, \rho), a), u)$.

Every SST $\mathcal{S}$ denotes a partial function $[\![\mathcal{S}]\!] : \Sigma^* \to \Delta^* \cup \{\emptyset\}$ where for any $u \in \Sigma^*$, we define

$$[\![\mathcal{S}]\!](u) = \begin{cases} \widehat{\rho'}(F_\mathcal{S}(q')) & \text{if } \delta^*((q^-, \rho^-), u) = (q', \rho') \\ & \text{and } q' \in \mathrm{dom}(F_\mathcal{S}) \\ \text{undefined} & \text{otherwise} \end{cases}$$

4.3 Tabulation

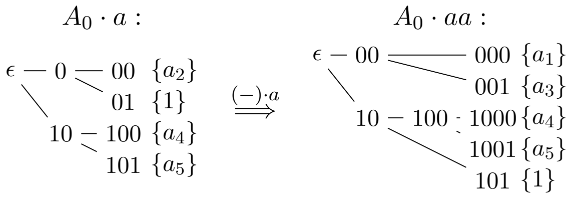
We need to come up with a representation of our streaming simulation algorithm as an SST with a designated register used for streaming output. Our representation needs to satisfy the property of being finite state as well as the property that the output register contains the output $p_1 p_2 ... p_i$ of Algorithm 1 after reading input symbol a_i. The latter requirement means that we must somehow statically encode the prefix structure of all potential outputs in the states of the SST, since SSTs cannot access the contents of registers. It turns out that this is possible by letting the states of the SST be equivalence classes of generalized state sets, where the equivalence relates state sets that agree on state ordering and prefix structure.

Trees We will call a prefix-free generalized state set A an *ordered tree* with node set

$$N_A = \{A(p) \wedge A(q) \mid p, q \in Q, A(p) \wedge A(q) \neq \emptyset\}.$$

Under this view, the leaves of A seen as a tree is the subset of nodes $L_A = \{A(q) \mid q \in Q, A(q) \neq \emptyset\} \subseteq N_A$, and the leaves are labeled by $A^{-1} : L_A \to 2^Q$. Since A is assumed prefix-free, we have for any nodes $x, y \in N_A$ that $x \preceq y$ if and only if there is a $z \in N_A$ such that $x = y \wedge z$. In this case x is called an *ancestor* of y and z, which in turn are called the *descendants* of x. Importantly, the root node of any (sub)tree is the longest common prefix of its descendants.

Example 1. We illustrate the tree interpretation as follows. Consider the oracle transducer from Figure 1. Let A_0 be the generalized state set that maps N_{main} to ϵ and every other state to $\emptyset$. Then the state sets $A_0 \cdot a$ and $A_0 \cdot aa$ can be seen as trees in the following way:



We will consider two generalized state sets to be equivalent if they are indistinguishable as ordered trees.

Definition 9 (Ordered tree isomorphism)**.** Let D_1, D_2 be totally ordered and let $A_1 \in S(D_1, Q)$ and $A_2 \in S(D_2, Q)$ be trees. An ordered tree isomorphism between A_1 and A_2 is a bijective map $h : N_{A_1} \to N_{A_2}$ such that for all $p, q \in Q$:

1. $h(A_1(p) \wedge A_1(q)) = A_2(p) \wedge A_2(q)$; and
2. $A_1(p) \leq A_1(q)$ if and only if $A_2(p) \leq A_2(q)$.

We write $h : A_1 \equiv A_2$ and say that A_1 and A_2 are *equivalent* when h is an ordered tree isomorphism between A_1 and A_2. Tree equivalence is preserved by the right action:

Proposition 5. *If $A \in (D_1, Q), B \in (D_2, Q)$ and $h : A \equiv B$ then for all $a \in \Sigma$, we have $A \cdot a \equiv B \cdot a$.*

Proof sketch. Since h is an order isomorphism and since A and B are prefix-free, we have for all $q \in Q$ exists $p_q \in Q$ and $y_q \in \Delta^*$ such that $(A \cdot a)(q) = A(p_q)y_q$ and $(B \cdot a)(q) = h(A(p_q))y_q$. Observe that for any $n \in N_{A \cdot a}$ there exists $q_1, q_2 \in Q$ such that

$$\begin{aligned} n &= (A \cdot a)(q_1) \wedge (A \cdot a)(q_2) \\ &= \begin{cases} A(q_1)(y_{q_1} \wedge y_{q_2}) & \text{if } A(q_1) = A(q_2) \\ A(q_1) \wedge A(q_2) & \text{otherwise} \end{cases} \end{aligned}$$

Furthermore, there does not exist $q_1, q_2, r_1, r_2 \in Q$ such that $A(q_1)(y_{q_1} \wedge y_{q_2}) = A(r_1) \wedge A(r_2)$, since that would imply that $A(q_1)$ is a prefix of $A(r_1)$ and $A(r_2)$. We define a map $h' : N_{A \cdot a} \to N_{B \cdot a}$ such that for all $q_1, q_2 \in Q$,

$$\begin{aligned} &h'((A \cdot a)(q_1) \wedge (A \cdot a)(q_2)) \\ &\qquad = \begin{cases} h(A(q_1)(y_{q_1} \wedge y_{q_2})) & \text{if } A(q_1) = A(q_2) \\ h(A(q_1) \wedge A(q_2)) & \text{otherwise.} \end{cases} \end{aligned}$$

This is a well-defined function by the previous observations, and a tree isomorphism by the fact that h is a tree isomorphism. □

Canonical representatives Call a generalized set $A \in S(D, Q)$ *canonical* if

1. $\mathrm{rng}(A)$ is prefix closed: if $y \in \mathrm{rng}(A)$ and $x \preceq y$ then $x \in \mathrm{rng}(A)$; and
2. $\mathrm{rng}(A)$ is downwards closed: if $x\ b \in \mathrm{rng}(A)$ for $b' < b$ then $xb' \in \mathrm{rng}(A)$ (for $b, b' \in \Delta$).

Write $\widetilde{S}(D, Q)$ for the subset of canonical trees. The set is finite, as every canonical tree A has a prefix closed node set, so the longest word in N_A is bounded by $|\mathrm{dom}(A)| - 1$ (the maximum depth of a tree with $|\mathrm{dom}(A)|$ leaves).

Any tree has a canonical representative:

Proposition 6. *For any set D and tree $A \in S(D, Q)$, there is a unique $C \in \widetilde{S}(\mathbb{N}, Q)$ with $A \equiv C$.*

As a consequence, there is a reduction map $[\cdot] : S(D, Q) \to \widetilde{S}(\mathbb{N}, Q)$ such that $A \equiv B$ if and only if $[A] = [B]$, implying that the quotient set $S(D, Q)/\equiv$ must be finite. Any $A \in S(D, Q)$ is thus canonically represented by a homomorphism $h_A : N_{[A]} \to N_A$ such that $A = h_A \circ [A]$.

In view of Proposition 5, this means that we can statically enumerate all possible trees up to tree isomorphism by computing with the canonical representatives. Any concrete tree reachable by the simulation algorithm is an instance of a canonical tree composed with a suitable homomorphism. An SST implementing the simulation algorithm can thus take the set of canonical trees as its states, and will then need to maintain the associated homomorphism via register updates.

Paths We need to represent tree homomorphisms using SST registers such that the effect of computing right actions on the underlying tree can be expressed as SST updates.

For a tree $A \in S(D, Q)$, any node $x \in N_A$ has a unique maximal decomposition $x = x_0x_1...x_n$ such that each $x_0x_1...x_i \in N_A$ for all $0 \leq i \leq n$. Intuitively, this reflects the full path from the root node to the node x, and we can define the map

$$\begin{aligned}\mathsf{path}_A &: N_A \to N_A^* \\ \mathsf{path}_A(x) &= (x_0, x_0x_1, ..., x_0x_1...x_n),\end{aligned}$$

which maps nodes to their maximal path decomposition (we use the tuple notation to distinguish between the two levels of monoids). In view of this and the fact that homomorphisms must preserve descendants, then for any homomorphism $h : A \equiv B$ there is a unique $\kappa_h : N_A \to N_B$ such that

$$h(x) = \kappa_h(t_0)\kappa_h(t_1) \cdots \kappa_h(t_n), \tag{1}$$

where $\mathsf{path}_A(x) = (t_0, t_1, ..., t_n)$. Intuitively, κ can be seen as a "differential" representation of h, representing the change of h between a node and its immediate ancestor. By viewing κ_h as a map $N_A \to D_B^*$ which extends uniquely to a monoid homomorphism $\widehat{\kappa_h} : N_A^* \to D_B^*$, we obtain $h = \widehat{\kappa_h} \circ \mathsf{path}_A$. Considering the unique isomorphism $h_A : [A] \equiv A$, write κ_A for the associated decomposition satisfying (1), and we thus have

$$A = \widehat{\kappa_A} \circ \mathsf{path}_{[A]} \circ [A] \tag{2}$$

The path-operator is easily seen to be a tree isomorphism since it preserves node ordering and prefix structure. That is, for any $A \in S(D, Q)$, we have $\mathsf{path}_A : A \equiv A^\sharp$ where $A^\sharp \in S(N_D, Q)$ is defined by $A^\sharp = \mathsf{path}_A \circ A$. Using this notation, (2) becomes

$$A = \widehat{\kappa_A} \circ [A]^\sharp, \tag{3}$$

SST construction We construct an SST implementing the FST simulation algorithm and sketch a proof of its correctness.

Theorem 1. *For any normalized prefix-free transducer $\mathcal{T} = (\Sigma, \Delta, Q, q^-, q^f, E)$, there is an SST $\mathcal{S}$ such that $[\![S]\!] = [\![T]\!]_{\leq}$.*

Proof. We define $\mathcal{S}$ as follows. Let A_0 be defined as in Algorithm 1, and observe that $A_0 \in \widetilde{S}(\mathbb{N}, Q)$. The states are the canonical trees labeled by Q:

$$\begin{aligned} Q_{\mathcal{S}} &= \{[A] \mid A \in S(\Delta, Q)\} \cup \{A_0\} \subseteq \widetilde{S}(\mathbb{N}, Q), \\ q^-_{\mathcal{S}}(q) &= A_0(q) \end{aligned}$$

The registers will be identified by canonical tree nodes:

$$X_{\mathcal{S}} = \bigcup \{N_C \mid C \in Q_{\mathcal{S}}\}.$$

The final output and the transition maps are given as follows:

$$\begin{aligned} F_{\mathcal{S}}(C) &= (C^\sharp \cdot \epsilon)(q^f), \\ \delta^1_{\mathcal{S}}(C, a) &= [C \cdot a], \\ \delta^2_{\mathcal{S}}(C, a, x) &= \begin{cases} \kappa_{C^\sharp \cdot a}(x) & \text{if } x \in N_{[C^\sharp \cdot a]} \\ \epsilon & \text{otherwise} \end{cases} \end{aligned}$$

We claim that $\mathcal{S}$ computes the same function as $\mathcal{T}$ under the functional semantics.

For $u \in \Sigma^*$ let (C_u, ρ_u) refer to the value $\delta^*_{\mathcal{S}}((q^-_{\mathcal{S}}, \rho^-), u) = (C_i, \rho_i)$. We show that for any $u \in \Sigma^*$, we have $\widehat{\rho_u} \circ (C^\sharp_u \cdot \epsilon) = A_0 \cdot u$.

Suppose that this holds. Then for any $u \in \Sigma^*$, we have by the above and Proposition 2 that $[\![S]\!](u) = \widehat{\rho_u}(F_{\mathcal{S}}(C_u)) = \widehat{\rho_u} \circ (C^\sharp_u \cdot \epsilon)(q^f) = (A_0 \cdot u)(q^f) = [\![\mathcal{T}]\!]_{\leq}(u)$.

Our claim follows as a special case of the following lemma. □

Lemma 1. *Let $A \in S(\Delta, Q)$ and $\rho : X_{\mathcal{S}} \to \Delta^*$ such that $A = \widehat{\rho} \circ [A]^\sharp$. Then for any $u \in \Sigma^+$ with $\delta^*_{\mathcal{S}}(([A], \rho), u) = (C, \rho')$ we have $\widehat{\rho'} \circ C^\sharp = A \cdot u$.*

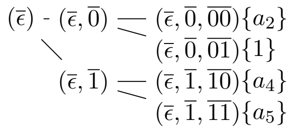
Proof. By induction on u. For $u = a$ we have $C = [[A] \cdot a] = [A \cdot a]$ and $\rho' = \widehat{\rho} \circ \kappa_{[A]^\sharp \cdot a}$. We can easily verify that $\widehat{\rho'} = \widehat{\rho} \circ \widehat{\kappa_{[A]^\sharp \cdot a}}$ so for any $q \in Q$,

$$\begin{aligned} \widehat{\rho'} \circ [A \cdot a]^\sharp(q) &= \widehat{\rho} \circ \widehat{\kappa_{[A]^\sharp \cdot a}} \circ [A \cdot a]^\sharp(q) \\ &= \widehat{\rho} \circ ([A]^\sharp \cdot a)(q) \\ &= \widehat{\rho}(\min\{[A]^\sharp(p)y \mid p \xrightarrow{a|y}_{\mathsf{np}} q \downarrow\}) \\ &= \min\{\widehat{\rho} \circ [A]^\sharp(p)y \mid p \xrightarrow{a|y}_{\mathsf{np}} q \downarrow\} \\ &= \min\{A(p)y \mid p \xrightarrow{a|y}_{\mathsf{np}} q \downarrow\} = (A \cdot a)(q) \end{aligned}$$

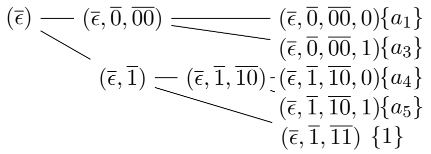
The second equality follows by observing that $A \equiv [A] \equiv [A]^\sharp$, so by Proposition 5, we have $A \cdot a \equiv [A]^\sharp \cdot a$ and thus $[A \cdot a] = [[A]^\sharp \cdot a]$. Therefore, $\widehat{\kappa_{[A]^\sharp \cdot a}} \circ [A \cdot a]^\sharp = \widehat{\kappa_{[A]^\sharp \cdot a}} \circ [[A]^\sharp \cdot a]^\sharp = [A]^\sharp \cdot a$ by using the identity (3). The fourth equality is justified by the fact that $[A]^\sharp(p) \leq [A]^\sharp(q)$ if and only if $A(p) \leq A(q)$.

For $u = au'$ where $u' \neq \epsilon$, we have $(C, \rho') = \delta^*_{\mathcal{S}}(([A \cdot a], \widehat{\rho} \circ \kappa_{[A]^\sharp \cdot a})$. By the previous argument we can apply the induction hypothesis, and we obtain $C = [(A \cdot a) \cdot u']$ and $\widehat{\rho}' \circ C^\sharp = (A \cdot a) \cdot u'$. The result then follows by Proposition 3. □

Example 2. We illustrate how the construction works by showing how Example 1 is implemented as an SST update between states $[A_0 \cdot a]$ and $[A \cdot aa]$. The register update is obtained by computing $\kappa_{[A_0 \cdot a]^\sharp \cdot a}$. The tree $[A_0 \cdot a]^\sharp$ looks as follows:


$(\bar{\epsilon})$
$(\bar{\epsilon}, \bar{0})$
$(\bar{\epsilon}, \bar{0}, \overline{00})\{a_2\}$
$(\bar{\epsilon}, \bar{0}, \overline{01})\{1\}$
$(\bar{\epsilon}, \bar{1})$
$(\bar{\epsilon}, \bar{1}, \overline{10})\{a_4\}$
$(\bar{\epsilon}, \bar{1}, \overline{11})\{a_5\}$

Recall that each node is a full path in the canonical tree $[A_0 \cdot a]$. The node names from $N_{[A_0 \cdot a]}$ are overlined and elements of the path monoid $N^*_{[A_0 \cdot a]}$ is written $(x_1, x_2, ...)$. The tree $[A_0 \cdot a]^\sharp \cdot a$ looks as follows:


$(\bar{\epsilon})$
$(\bar{\epsilon}, \bar{0}, \overline{00})$
$(\bar{\epsilon}, \bar{0}, \overline{00}, 0)\{a_1\}$
$(\bar{\epsilon}, \bar{0}, \overline{00}, 1)\{a_3\}$
$(\bar{\epsilon}, \bar{1})$
$(\bar{\epsilon}, \bar{1}, \overline{10})$
$(\bar{\epsilon}, \bar{1}, \overline{10}, 0)\{a_4\}$
$(\bar{\epsilon}, \bar{1}, \overline{10}, 1)\{a_5\}$
$(\bar{\epsilon}, \bar{1}, \overline{11})\ \{1\}$

Note that symbols that are not overlined are output symbols from Δ. The map $\kappa' = \kappa_{[A_0 \cdot a]^\sharp \cdot a} : N_{[A_0 \cdot aa]} \to (N_{[A_0 \cdot a]} \cup \Delta)^*$ gives us the relevant SST update strings:

$$\begin{array}{rlrlrl}
\kappa'(\bar{\epsilon}) = (\bar{\epsilon}) & & \kappa'(\bar{0}) = (\bar{0}, \overline{00}) & & \kappa'(\overline{00}) = 0 & \\
\kappa'(\overline{01}) = 1 & & \kappa'(\bar{1}) = (\bar{1}) & & \kappa'(\overline{10}) = (\overline{10}) & \\
\kappa'(\overline{100}) = 0 & & \kappa'(\overline{101}) = 1 & & \kappa'(\overline{11}) = (\overline{11}) &
\end{array}$$

The full construction of an SST from the oracle transducer in Figure 1 can be seen in Figure 2.

5 Implementation

Our implementation compiles a Kleenex program to machine code by implementing the transducer constructions described in the earlier sections. We have also implemented several optimizations to decrease the size of the generated SSTs and improve the performance of the generated code. We will briefly describe these in the following section, and we note that they are all orthogonal to the underlying principles behind our compilation.

The possible compilation paths of our implementation can be seen in Fig. 3.

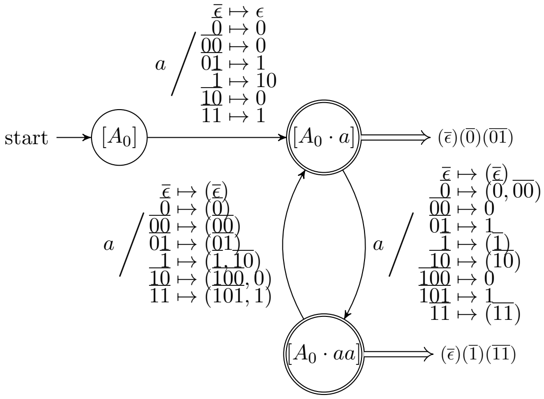

start
$[A_0]$
$[A_0 \cdot a]$
$[A_0 \cdot aa]$
a
$\overline{\epsilon} \mapsto \epsilon$
$\overline{0} \mapsto 0$
$\overline{00} \mapsto 0$
$\overline{01} \mapsto 1$
$\overline{1} \mapsto 10$
$\overline{10} \mapsto 0$
$\overline{11} \mapsto 1$
$(\overline{\epsilon})(\overline{0})(\overline{01})$
a
$\overline{\epsilon} \mapsto (\overline{\epsilon})$
$\overline{0} \mapsto (\overline{0})$
$\overline{00} \mapsto (\overline{00})$
$\overline{01} \mapsto (\overline{01})$
$\overline{1} \mapsto (\overline{1}, \overline{10})$
$\overline{10} \mapsto (\overline{100}, 0)$
$11 \mapsto (101, 1)$
a
$\overline{\epsilon} \mapsto (\overline{\epsilon})$
$\overline{0} \mapsto (0, \overline{00})$
$\overline{00} \mapsto 0$
$\overline{01} \mapsto 1$
$\overline{1} \mapsto (\overline{1})$
$\overline{10} \mapsto (\overline{10})$
$\overline{100} \mapsto 0$
$\overline{101} \mapsto 1$
$11 \mapsto (\overline{11})$
$(\overline{\epsilon})(\overline{1})(\overline{11})$

Figure 2: Example of SST computing the same function as the oracle transducer in Figure 1. Each transition is tagged by a register update, and the nodes of the canonical tree identifying the destination state make up the registers. The wide arrows exiting the accepting states indicate the final output string. Note that this always includes the root variable ($\overline{\epsilon}$) which thus acts as an interface for streaming output (although for this particular example, nothing can output until the end of the input).

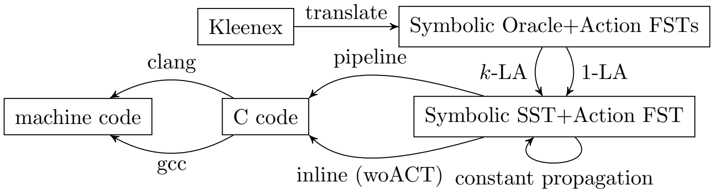


Figure 3: Compilation paths. 1-LA is symbolic SST construction with single-symbol transitions; k-LA is construction of SST with up to k symbols of lookahead for some k determined by the program. The “pipeline” translation path indicates that the resulting program keeps the oracle SST and action FST separate, with data being piped from the SST to the FST at runtime. The “inline” path indicates that the action FST is fused into the oracle SST.

5.1 Transducer pipeline

It is possible to chain together several Kleenex programs in a pipeline, letting the output of one serve as the input of the next. This can for example be used to strip unwanted characters before performing a transformation. By using the optional *pipeline pragma*, `start:` t_1 `>>` ... `>>` t_n, a programmer can specify that the entry point is t_1 and that the output should be chained together as specified, with the final output being that of t_n. The implementation does this by spawning a process for each transducer and setting up UNIX pipes between them.

5.2 Inlining the action transducer

When we have constructed the oracle SST we end up with two deterministic machines which need to be composed. We can either do this at runtime, piping the output of the oracle SST into the action FST, or we can apply a form of deforestation to inline the outuput of the action FST directly in the SST (this is straightforward since the action FST is deterministic by construction). The former approach is advantageous if the Kleenex program produces a lot of output and is highly nondeterministic.

5.3 Constant propagation

The SSTs generated by our construction contains quite a lot of trivial register updates which can be eliminated in order to achieve better run-time efficiency. Consider the SST in Fig. 2, where all registers but $(\overline{0})$ and $(\overline{1})$ are easily seen to have a constant known value in each state. Eliminating the redundant registers means that we only have to maintain two registers at run-time.

We achieve this by constant propagation: computing reaching definitions by solving a set of data-flow constraints (see e.g. [**?**]).

5.4 Symbolic representation

Text transformation programs often contain idioms which have a rather redundant representation as pure transducers. A program might for example match against a whole range of characters and proceed in the same way regardless of which one was matched. This will, however, lead to a transition for each concrete character in the generated FST, even though all transitions have the same source and destination states.

A more succinct representation can be obtained by using a symbolic representation of the transition relation by introducing transitions whose input labels are *predicates*, and whose output labels are *terms* indexed by input symbols. Replacing input labels with predicates has been described first described by Watson [**?**]. Such symbolic transducers have been developed further and have recently received quite a bit of attention, with applications in verification and verifiable string transformations [**?**, **?**, **?**, **?**].

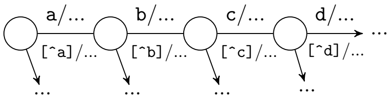
Our implementation of Kleenex uses a symbolic representation for basic ranges of symbols in order to get rid of most redundancies. The simulation algorithm and the SST construction can be generalized to the symbolic case without altering the fundamental structure, so we have elided the details of this optimization. We refer the reader to the cited literature for the technical details of symbolic transducers.

5.5 Finite lookahead

A common pattern in Kleenex programs are definitions of the form

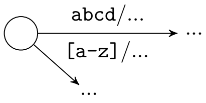
```
token := ~/abcd/ commonCase | ~/[a-z]+/ fallback
```

that is, a specific pattern appearing with higher priority than a more general fallback pattern. Patterns of this form will result in (symbolic) SSTs containing the following kind of structure:



The primary case and the fallback pattern are simulated in lockstep, and in each state there is a transition for when the common case fails after reading 0, 1, 2, etc. symbols.

If the SST was able to look more than one symbol ahead before determining the next state, we would be able to tabulate a much coarser set of simulation states and do away with the fine-grained interleaving. For the above example, we would like a transition structure like the following:



If the first four symbols of the input are `abcd`, the upper transition is taken. If this is not the case, but the first symbol is `a`, then the lower transition is taken. The idea is that any string successfully matched by the primary case will satisfy the test `abcd`, so if the transition with `[a-z]` is taken, then the FST states corresponding to the primary case can be removed from the generalized state set and tabulation can continue with a simpler simulation state.

The semantics of SSTs with lookahead are still deterministic despite the seeming overlap of patterns, as the model requires that any pair of tests are either disjoint (no string will satisfy both at the same time), or one test is completely contained in another (if a string satisfies the first test, it also satisfies the second). This restriction gives a total order between tests, specifying their priority—the most specific test must be tried first.

6 Benchmarks

We have run comparisons with different combinations of the following tools:

RE2, Google's automata-based regular expression C++ library [**?**].
RE2J, a recent re-implementation of RE2 in Java [**?**].
GNU `AWK`, GNU `grep`, and GNU `sed`, programming languages and tools for text processing and extraction [**?**].
Oniglib, a regular expression library written in C++ with support for different character encodings [**?**].
Ragel, a finite state machine compiler with multiple language backends [**?**].

In addition, we implemented test programs using the standard regular expression libraries in the scripting languages Perl [**?**], Python [**?**], and Tcl [**?**].

Meaning of plot labels Kleenex plot labels indicate the compilation path, and follow the format `[<0|3>[-la] | woACT] [clang|gcc]`. `0/3` indicates whether constant propagation was disabled/enabled. `la` indicates whether lookahead was enabled. `clang/gcc` indicates which C compiler was used. The last part indicates that custom register updates are disabled, in which case we generate a single fused SST as described in 6.3. These are only run with constant propagation and lookahead enabled.

Experimental setup The benchmark machine runs Linux, has 32 GB RAM and an eight-core Intel Xeon E3-1276 3.6 GHz CPU with 256 KB L2 cache and 8 MB L3 cache. Each benchmark program was run 15 times, after first doing two warm-up rounds. Version numbers of libraries, etc. are included in the appendix. All C and C++ files have been compiled with `-O3`.

Difference between Kleenex and the other implementations Unless otherwise stated, the structure of all the non-Kleenex implementations is a loop that reads input line by line and applies an action to the line. Hence, in these implementations there is an interplay between the regular expression library used and the external language, e.g., RE2 and C++. In Kleenex, line breaks do not carry any special significance, so the multi-line programs can be formulated entirely within Kleenex.

Ragel optimization levels Ragel is compiled with three different optimization levels: T1, F1, and G2. "T1" and "F1" means that the generated C code should be based on a lookup-table, and "G2" means that it should be based on C `goto` statements.

Kleenex compilation timeout On some plots, some versions of the Kleenex programs are not included. This is because the C compiler has timed out (after 30 seconds). As we fully determinize the transducers, the resulting C code can explode in some cases. This is a an area for future research.

6.1 Baseline

The following three programs are intended to give a baseline impression of the performance of Kleenex programs.

`flip_ab` The program `flip_ab` swaps "a"s and "b"s on all its input lines. In Kleenex it looks like this:

```
main := ("b" ~/a/ | "a" ~/b/ | /\n/)*
```

We made a corresponding implementation with Ragel, using a `while`-loop in C to get each new input line and feed it to the automaton code generated by Ragel.

Implementing this functionality with regular expression libraries in the other tools would be an unnatural use of them, so we have not measured those.

The performance of the two implementations run on input with an average line length of 1000 characters is shown in Figs. 4.

`patho2` The program `patho2` forces Kleenex to wait until the very last character of each line has been read before it can produce any output:

```
main := ((~/[a-z]*a/ | /[a-z]*b/)? /\n/)+
```

In this benchmark, the constant propagation makes a big difference, as Fig. 5 shows. Due to the high degree of interleaving and the lack of keywords, in this program the look-ahead optimization reduces overall performance.

This benchmark was not run with Ragel because Ragel requires the programmer to do all disambiguation manually when writing the program; the C code that Ragel generates does not handle ambiguity in a predictable way.

6.2 Rewriting

Thousand separators The following Kleenex program inserts thousand separators in a sequence of digits:

```
main  := (num /\n/)*
num   := digit{1,3} ("," digit{3})*
digit := /[0-9]/
```

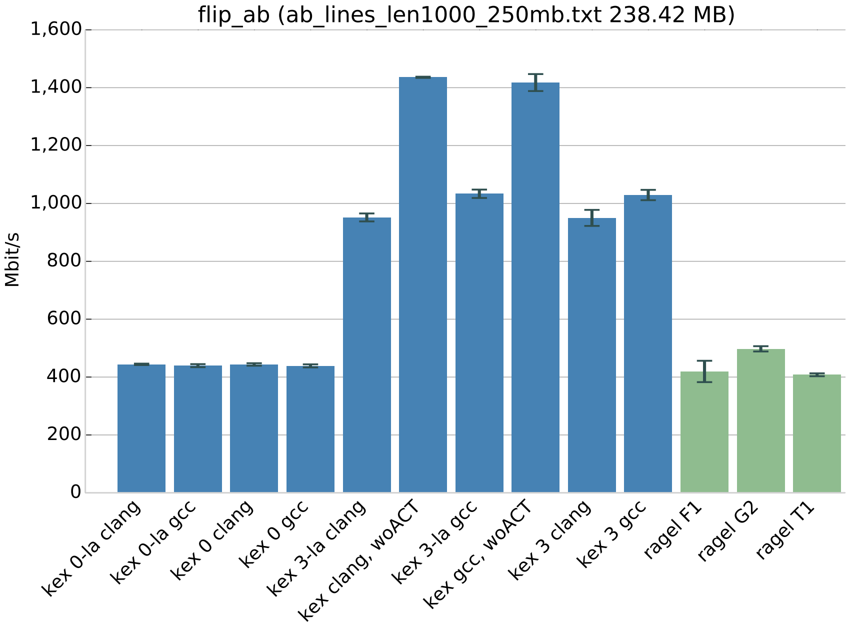


Figure 4: `flip_ab` run on lines with average length 1000.

We evaluated the Kleenex implementation along with three other implementations using Perl, and Python. The performance can be seen in Fig. 6. Both Perl and Python are significantly slower than all of the Kleenex implementations; this is a problem that is tricky to formulate with normal regular expressions (unless one reads the input right-to-left).

CSV rewriting The program `csv_project3` deletes columns two and five from a CSV file:

```
main := (row /\n/)*
col  := /[^,\n]*/
row  := ~(col /,/) col "\t" ~/,/ ~(col /,/)
        ~(col /,/) col ~/,/      ~col
```

Various specialized tools exist that can handle this transformation are included in Fig. 7; GNU `cut` is a command that splits its input on certain characters, and GNU `AWK` has built-in support for this type of transformation.

Apart from `cut`, which is really fast for its own use-case, the Kleenex implementation is the fastest. The performance of Ragel is slightly lower, but this is likely due to the way the implementation produces output: In a Kleenex program, output strings are automatically put in an output buffer

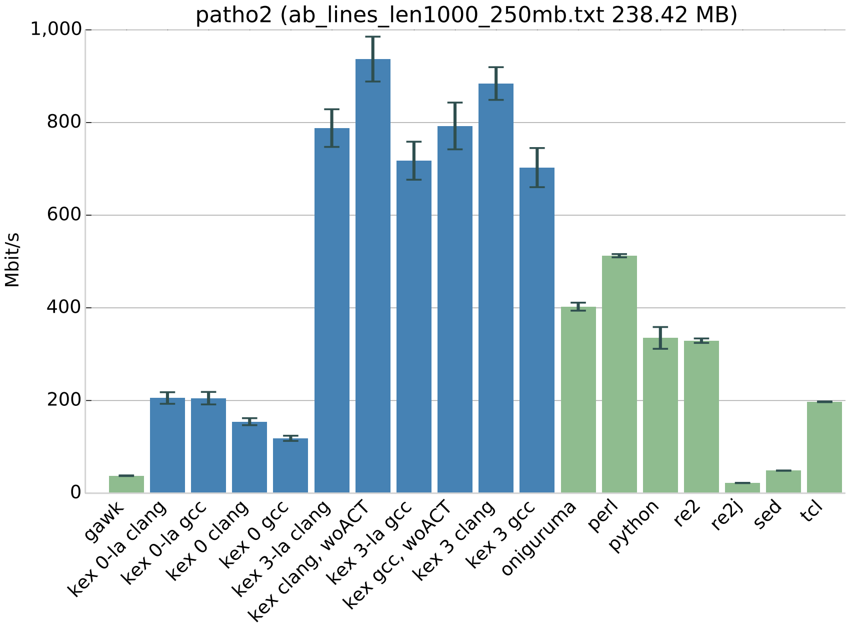


Figure 5: `patho2` run on lines with average length 1000.

which is flushed routinely, whereas a programmer has to manually handle buffering when writing a Ragel program.

IRC protocol handling The following Kleenex program parses the IRC protocol as specified in RFC 2812.[4] It follows roughly the output style described in part 2.3.1. Note that the Kleenex source code and the BNF grammar in the RFC are almost identical. Fig. 8 shows the throughput on 250 MiB data.

```
main := (message | "Malformed line: " /[^\r\n]*\r?\n/)*
message := (~/:/ "Prefix: " prefix "\n"  ~/ /)?
           "Command: " command "\n"
           "Parameters: " params? "\n"
           ~crlf
command := letter+ | digit{3}
prefix := servername
        | nickname ((/!/ user)? /@/ host )?
user := /[^\n\r @]/+ // Missing \x00
middle := nospcrlfcl ( /:/ | nospcrlfcl )*
params := (~/ / middle ", "){,14} ( ~/ :/ trailing )?
        | ( ~/ / middle ){14} ( / / /:/?  trailing )?
```

[4] https://tools.ietf.org/html/rfc2812

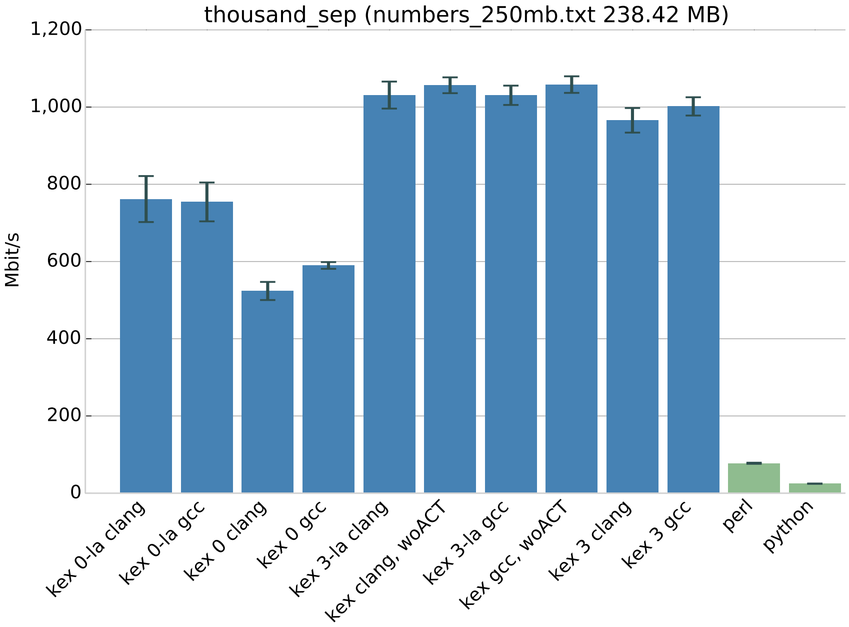


Figure 6: Inserting thousand separators on random numbers with average length 1000.

```
trailing := (/:/ | / / | nospcrlfcl)*
nickname := (letter | special)
            (letter | special | digit){,10}
host := hostname | hostaddr
servername := hostname
hostname := shortname ( /\./ shortname)*
hostaddr := ip4addr
shortname := (letter | digit) (letter | digit | /-/)*
             (letter | digit)*
ip4addr := (digit{1,3} /\./ ){3} digit{1,3}
```

6.3 With or without action-separation

One can choose to use the machine resulting in combining the oracle and the action machine when compiling Kleenex. Doing so results in only one process doing both the disambiguation and outputting, which in some cases is faster and in other slower. Figs. 7, 9, and 11 illustrate both situations. It depends on the structure of the problem whether it pays off to split up the work in two processes; if all the work happens in the oracle and the action machine nearly does not do anything, then the added overhead incurred by the process context switches becomes noticeable. On the other hand, in

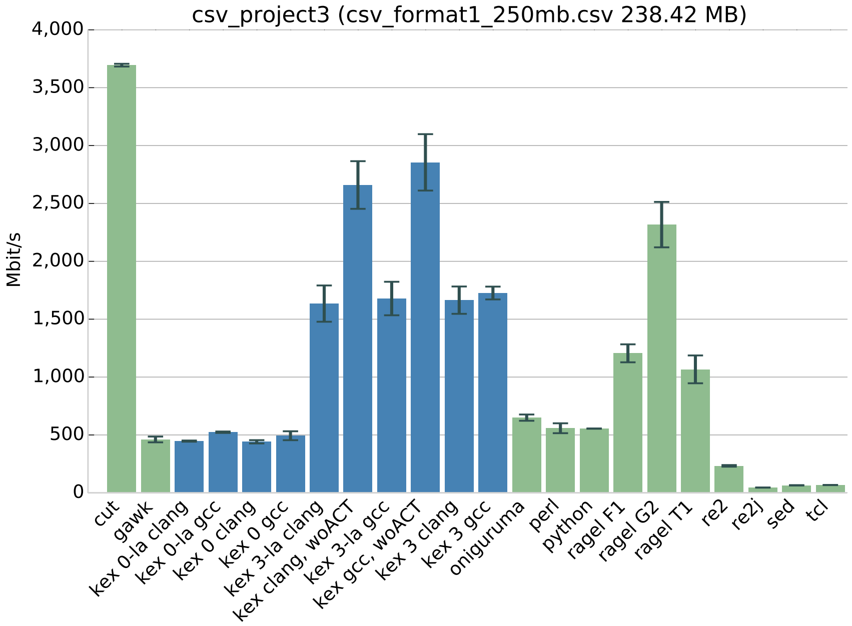


Figure 7: `csv_project3` reads in a CSV file with six columns and outputs columns two and five. "gawk" is GNU `AWK` that uses the native `AWK` way of splitting up lines. "cut" is a command from GNU coreutils that splits up lines.

cases where both machines do much work, the fact that two CPU cores can be utilized speeds up the program. This would be more likely if Kleenex had support for actions which could perform arbitrary computation, e.g. in the form of embedded C code.

7 Use cases

In this section we will briefly touch upon various interesting use cases for Kleenex.

JSON logs to SQL We have implemented a Kleenex program (code in Appendix) that transforms a JSON log file into an SQL insert statement. The program works on the logs provided by Issuu.[5]

The Ragel version we implemented outperforms Kleenex by about 50% (Fig. 9), indicating that further optimizations of our SST construction should be possible.

[5] The line-based data set consists of 30 compressed parts and part one is available from `http://labs.issuu.com/anodataset/2014-03-1.json.xz`

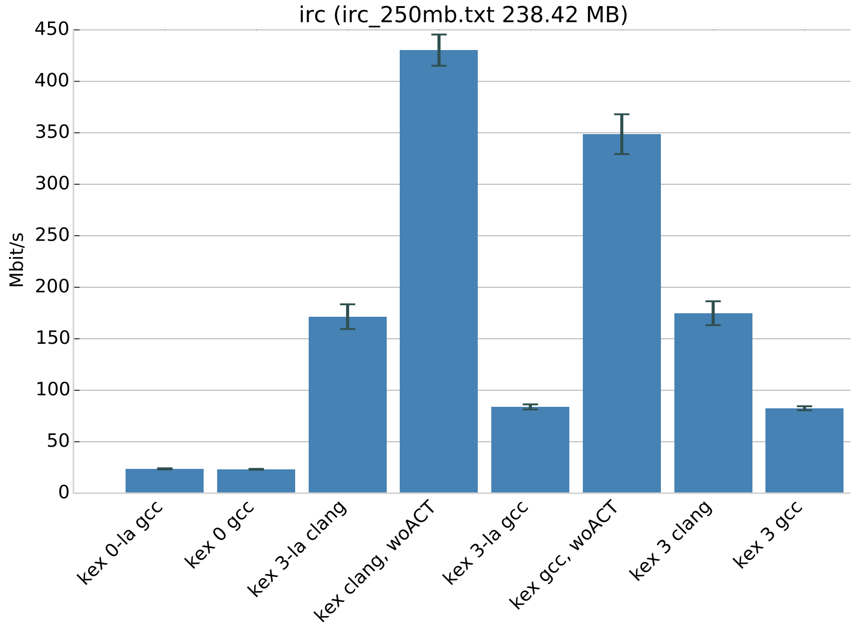


Figure 8: Throughput when parsing 250 MiB random IRC data.

Apache CLF to JSON The Kleenex program below rewrites Apache CLF[6] log files into a list of JSON records:

```
main := "[" loglines? "]\n"
loglines := (logline "," /\n/)* logline /\n/
logline := "{" host ~sep ~userid ~sep ~authuser sep
               timestamp sep request sep code sep
               bytes sep referer sep useragent "}"
host := "\"host\":\"" ip "\""
userid := "\"user\":\"" rfc1413 "\""
authuser := "\"authuser\":\"" /[^ \n]+/ "\""
timestamp := "\"date\":\"" ~/\[/ /[^\n\]]+/ ~/]/ "\""
request := "\"request\":" quotedString
code := "\"status\":\"" integer "\""
bytes := "\"size\":\"" (integer | /-/) "\""
referer := "\"url\":" quotedString
useragent := "\"agent\":" quotedString
ws := /[\t ]+/
sep := "," ~ws
quotedString := /"([^"\n]|\\")*"/
integer := /[0-9]+/
ip := integer (/\./ integer){3}
```

[6] https://httpd.apache.org/docs/trunk/logs.html#common

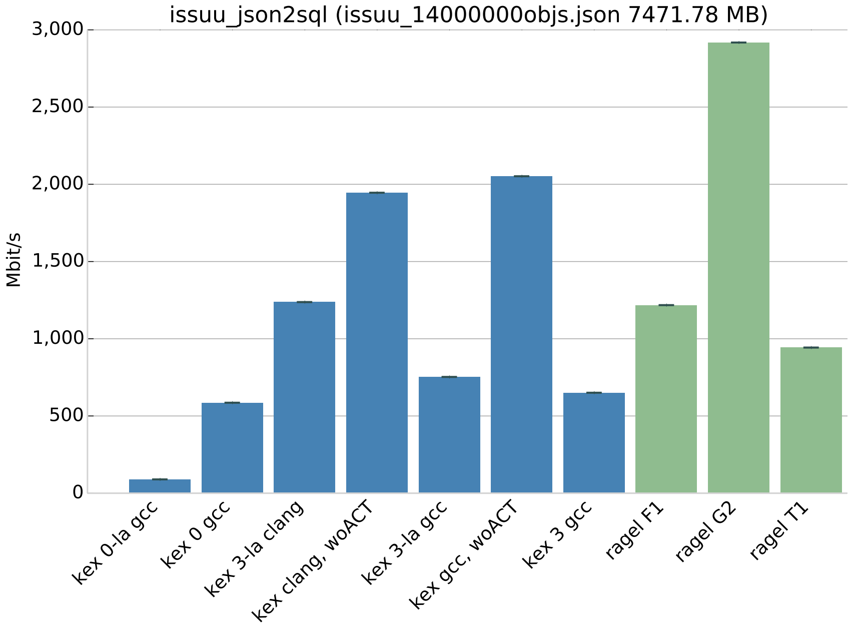


Figure 9: The speeds of transforming JSON objects to SQL INSERT statements using Ragel and Kleenex.

```
rfc1413 := /-/
```

This is a re-implementation of a Ragel program.[7] Fig. 10 shows the benchmark results. The versions compiled with clang are not included, as the compilation timed out after 30 seconds. Curiously, the non-optimized Kleenex program is the fastest in this case.

ISO date/time objects to JSON Inspired by an example in [**?**], the program `iso_datetime_to_json` (code in Appendix) converts date and time stamps in an ISO standard format to a JSON object. Fig. 11 shows the performance.

URL parsing Kleenex allows one to naturally follow the URL specification given in RFC1738.[8] We implemented a URL parser by directly following the BNF-grammar in the RFC; its code can be found in the Appendix.

Syntax highlighting Kleenex can used to write syntax highlighters; in fact, the Kleenex syntax in this paper was highlighted with a Kleenex pro-

[7] `https://engineering.emcien.com/2013/04/5-building-tokenizers-with-ragel`
[8] `http://www.ietf.org/rfc/rfc1738.txt`

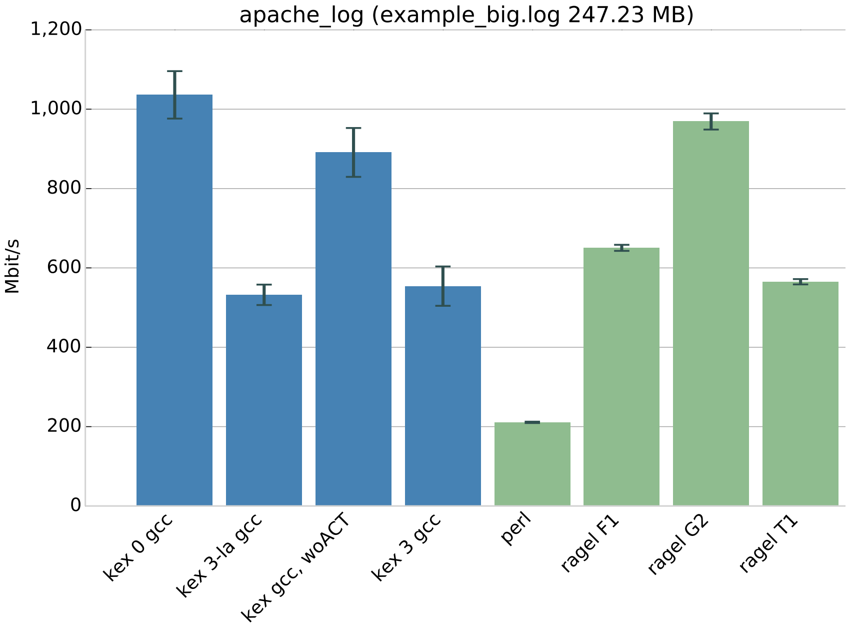


Figure 10: Speed of the conversion from the Apache Common Log Format to JSON.

gram. The code for a version that emits ANSI color codes is included in the Appendix.

HTML comments The following Kleenex program finds HTML comments with basic formatting commands and renders them in HTML after the comment. For example, `<!-- doc:  *Hello* world -->` becomes `<!-- doc:  *Hello* world --><div> <b>Hello</b> world </div>`.

```
main := (comment | /./)*
comment := /<!-- doc:/ clear doc* !orig /-->/
           "<div>" !render "</div>"
doc := ~/\*/ t@/[^*]*/ ~/\*/
       [ orig += "*" t "*" ] [ render += "<b>" t "</b>" ]
     | t@/./ [ orig += t ] [ render += t ]
clear := [ orig  <- "" ] [ render <- "" ]
```

8 Related Work

We discuss related work in the context of current and future work.

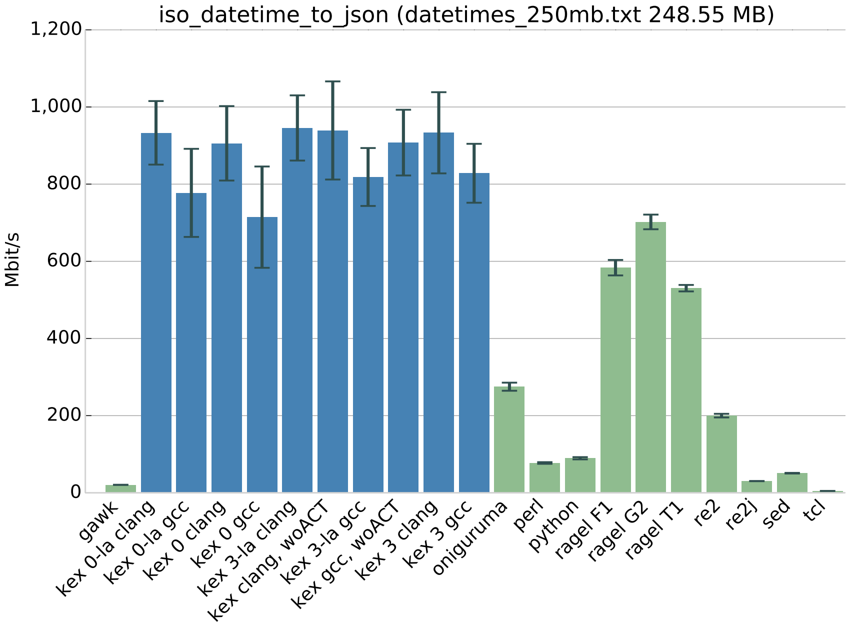


Figure 11: The performance of the conversion of ISO time stamps into JSON format.

8.1 Regular expression matching

Regular expression *matching* has different meanings in the literature.

For *acceptance testing*, which corresponds to classical automata theory, Bille and Thorup [**?**] improve on Myers' [**?**] log-factor improved RE-membership testing of classical NFA-simulation, based on tabling. They design an $O(kn)$ algorithm [**?**] with word-level parallelism, where $k \leq m$ is number of strings occurring in an RE. The tabling technique may be promising in practice; the algorithms have not been implemented and evaluated empirically, though.

In *subgroup matching* as in PCRE [**?**], an input is not only classified as accepting or not, but a substring is returned for each sub-RE in an RE designated to be of interest. Subgroup matching is often implemented by backtracking over alternatives, which yields the greedy match.[9] It may result in exponential-time behavior, however. Consequently, considerable human effort is expended to engineer REs to perform well. REs resulting in exponential run-time behavior are used in algorithmic attacks, leading to proposals for countermeasures to such attacks by classifying REs with

[9] Committing to the left alternative before checking that the remainder of the input is accepted is the essence of parsing expression grammars [**?**].

slow backtracking performance [?, ?], where the countermeasures in turn appear to be attackable. Even in the absence of inherently hard matching with backreferences [?], backtracking implementations with avoidable performance blow-ups are amazingly wide-spread. This may be due to a combination of their good best-case performance and PCRE-embellishments driven by use cases. Some submatch libraries with guaranteed worst-case linear-time performance, notably RE2 [?], are making inroads, however. Myers, Oliva and Guimaraes [?] and Okui, Suzuki [?] describe a $O(mn)$, respectively $O(m^2 n)$ POSIX-disambiguated matching algorithms. Sulzmann and Lu [?] use Brzozowski [?] and Antimirov derivatives [?] for Perl-style subgroup matching for greedy and POSIX disambiguation.

Full RE *parsing* generalizes submatching: it returns a list of matches for each Kleene-star, also for nested ones. Kearns [?], Frisch and Cardelli [?] devise 3-pass linear-time *greedy* RE parsing; they require 2 passes over the input, the first consisting of reversing the entire input, before generating output in the third pass. Grathwohl, Henglein, Nielsen, Rasmussen devise a two-pass [?] and an optimally streaming [?] greedy regular expression parsing algorithm. Streaming guarantees that line-by-line RE matching can be coded as a single RE matching problem. Sulzman and Lu [?] remark that POSIX is notoriously difficult to implement correctly and show how to use Brzozowski derivatives [?] for POSIX RE parsing;

There are specialized RE matching tools and techniques too numerous to review comprehensively. We mention a few employing automaton optimization techniques applicable to Kleenex, but presently unexplored. Yang, Manadhata, Horne, Rao, Ganapathy [?] propose an OBDD representation for subgroup matching and apply it to intrusion detection REs; the cycle counts per byte appear a bit high, but are reported to be competitive with RE2. Sidhu and Prasanna [?] implement NFAs directly on an FPGA, essentially performing NFA-simulation in parallel; it outperforms GNU `grep`. Brodie, Taylor, Cytron [?] construct a multistride DFA, which processes multiple input symbols in parallel, and devise a compressed implementation on stock FPGA, also achieving very high throughput rates. Likewise, Ziria employs tabled multistriding to achieve high throughput [?]. Navarro and Raffinot [?], show how to code DFAs compactly for efficient simulation.

8.2 Ambiguity

REs may be ambiguous, which is irrelevant for acceptance testing, but problematic for submatching and parsing since the output depends on which amongst possibly multiple matches is to be returned. Brüggemann-Klein [?] provides an efficient $O(m^2)$ RE ambiguity testing algorithm. Vansummeren [?] illustrates differences between POSIX, first/longest and greedy matches. Colcombet [?] analyzes notions of (non)determinism of automata.

8.3 Transducers

From RE parsing it is a surprisingly short distance to the implementation of arbitrary nondeterministic finite state transducers (NFSTs) [?, ?]. In contrast to the situation for *automata*, nondeterministic transducers are strictly more powerful than deterministic transducers; this, together with observable ambiguity, highlights why RE parsing is more challenging than RE acceptance testing.

As we have seen, efficient RE parsing algorithms operate on arbitrary NFAs, not only those corresponding to REs. Indeed, REs are not a particularly convenient or compact way of specifying regular languages: they can be represented by *certain* small NFAs with low tree-width, but may be inherently quadratically bigger even for DFAs [?, Theorem 23]. This is why Kleenex employs context-free grammars restricted to denote regular languages, with embedded output actions, to denote NFSTs.

We have shown that NFSTs, in particular unambiguous NFSTs, can be implemented by a *subclass* of streaming string transducers (SSTs). SSTs extensionally correspond to regular transductions, functions implementable by 2-way deterministic finite-state transducers [?], MSO-definable string transductions [?] and a combinator language analogous to regular expressions [?]. The implementation techniques used in Kleenex appear to be directly applicable to all SSTs, not just the ones corresponding to NFSTs.

Allender and Mertz [?] show that the functions computable by register automata, which generalize output strings to arbitrary monoids, are in NC and thus inherently parallelizable. This is achievable by performing relational NFST-composition by matrix multiplication on the matrix representation of NFSTs [?], which can be performed by parallel reduction. This is tantamount to running an NFST from all states, not just the input state, on input string fragments. Mytkowicz, Musuvathi, Schulte [?] observe that there is often a small set of cut states sufficient to run each NFST. This promises to be an interesting parallel harness for a suitably adapted Kleenex implementation running on fragments of very large inputs.

Veanes, Molnar, Mytkowics [?] employ symbolic transducers [?, ?, ?] and a data-parallel intermediate language in the implementation of BEK for multicore execution.

9 Conclusions

We have presented Kleenex, a convenient language for specifying nondeterministic finite state transducers; and its compilation to machine code representations of streaming state transducers, which emit the output

Kleenex is comparatively expressive and performs consistently well—for complex regular expressions with nontrivial amounts of output almost always better in the evaluated use cases—vis-à-vis text processing tools such as RE2, Ragel, `grep`, `AWK`, `sed` and RE-libraries of Perl, Python and Tcl.

We believe Kleenex's clean semantics, streaming optimality, algorithmic generality, worst-case guarantees and absence of tricky code and special casing provide a useful basis for

- extensions to deterministic visible push-down automata, restricted versions of backreferences and approximate/probabilistic matching;
- known, but so far unexplored optimizations, such as multicharacter input processing, automata minimization and symbolic representation, hybrid NFST-simulation/SST-construction (analogous to NFA-simulation with NFA-state set memoization to implement on-demand DFA-construction);
- massively parallel (log-depth, linear work) input processing.

Acknowledgements

We thank Issuu for releasing their data set to the research community. This work has been partially supported by The Danish Council for Independent Research under Project 11-106278, "Kleene Meets Church: Regular Expressions and Types". The order of authors is insignificant; please list all authors—or none—when citing this paper.

Moldable Applications on Multi-Core Servers: Active Resource Management instead of Passive Resource Administration

Clemens Grelck

University of Amsterdam
Informatics Institute
System and Network Engineering Lab
Science Park 904
1098XH Amsterdam, Netherlands
c.grelck@uva.nl

Abstract. Malleable applications are programs that can, in principle, run with varying numbers of threads and thus on varying numbers of cores of a mult-core parallel system. Malleability is characteristic for many programming models from data-parallel to divide-and-conquer and streaming data flow where the actual amount of concurrency is application and data dependent and varies over time while the runtime system maps the actual concurrency to fixed number of kernel threads / cores. We argue that such a fixed choice of kernel threads is suboptimal in two scenarios. Firstly, an application may temporarily expose less concurrency than the underlying hardware offers. In this case the cores waste energy. Secondly, the number of hardware cores effectively available to an application may dynamically change in multi-application and/or multi-user environments. This leads to an over-subscription of the available hardware by individual applications, costly time scheduling by the operating system and, as a consequence, to both waste of energy and loss of performance.

We propose an active resource management service developed in the context of the data parallel array language SAC and the streaming macro data flow coordination language S-Net. Both languages are examples of malleable runtime systems where the set of resources could be changed dynamically without affecting the consistency of a running application. A system-wide resource management service controls the computing resources and assigns them to applications on the basis of dynamically changing intra-application requirements as well as on dynamically changing inter-application scenarios in a near-optimal way.

1 Introduction

Malleable applications are programs that can, in principle, run with varying numbers of threads and thus on varying numbers of cores of a mult-core parallel system. Malleability is a characteristic feature of many parallel runtime systems.

For example, in data-parallel applications the number of iterations of a parallelised loop and thus the available concurrency typically exceeds the total number of cores in a system by several if not many orders of magnitude. Consequently, data-parallel applications typically scale down the structurally available concurrency in the application to the actually available concurrency of the execution platform. This is done by applying one of several available loop scheduling techniques, such as block scheduling, cyclic scheduling, block-cyclic scheduling, (guided) self scheduling or data locality aware variations of them.

The same even compiled binary application can in principle and within certain limits run on any number of cores. Typically, however, the number of cores/threads used is provided at application start through a command line parameter or an environment variable and then remains as set throughout the entire application life time. Dynamic malleability is usually not exploited. Common examples of such data-parallel runtime systems are OpenMP[1] or our own functional data-parallel array programming language Single Assignment C [2, 3].

The principle of malleable applications that do not exploit this property dynamically is not at all limited to the data-parallel scenario. In divide-and-conquer style applications written for instance in modern versions of OpenMP[4] using explicit task parallelism or in Cilk[5]. In either case the divide-and-conquer style parallelism, in beneficial scenarios, just like the data parallel approach exposes much higher levels of concurrency than general-purpose multi-core systems can exploit. The solution here in one way or another is to employ a fixed number of worker threads and work stealing techniques to balance the intra-application workload.

As a last example we mention streaming applications as for instance written in the declarative coordination language S-Net [6, 7]. S-Net defines the coordination behaviour of networks of asynchronous, stateless components and their orderly interconnection via typed streams. S-Net achieves a near-complete separation of concerns between the engineering of sequential application building blocks (i.e. *application engineering)* and the composition or orchestration of these building blocks to form a parallel application (i.e. *concurrency engineering).* S-Net effectively implements a macro data flow model where components represent non-trivial computations. Again the level of concurrency is not determined by the S-Net streaming application, but instead by characteristics of individual program runs. The S-Net runtime system [8] effectively maps the available concurrency to a number of threads that is determined upon program startup by the user and then remains fixed throughout the application's runtime.

To summarise, it is common across a wide range of concurrent programming models to expose concurrency to the underlying compiler and runtime system tool chain on a certain level of granularity that typically is finer-grained than what the execution platform effectively offers. Among others this choice is motivated by the fact that in many scenarios the exact properties and characteristics of the to be used execution platform are not known at compile time. The common solution is to map down the finer-grained concurrency exposed by some application to a fixed set of kernel worker threads launched at program startup and mapped to the actual computing resources such as processors (sockets), cores and hyperthreads (hardware execution contexts) by the operating system.

This immediately raises the question as to how many such worker threads to use. In practice, two solutions prevail: either an application determines the actual execution machinery it runs on and launches as many worker threads as it finds hardware execution units (the greedy approach) or an application simply asks the user (the clueless approach). The latter, of course, provides ample opportunity for exprimentation, but in this work we focus on the non-expert user who simply aims at making good use of available resources without extra effort.

We argue that both approaches are undesirable for a number of reasons. As soon as the user of an application is not its programmer at the same time, he or she may not be able to make an educated choice, in particular as the number of cores continues to rise and, thus, the design space. Furthermore, any fixed number of worker threads used throughout a program run is suboptimal for two reasons.

Firstly, we waste energy for operating all computing resources initially chosen as soon as the application effectively exposes less concurrency in certain phases of the program execution. Both the divide-and-conquer as well as the streaming model of parallel program organisation are characterised by ramp-up and fade-out phases of concurrency. In non-trivial applications it is fairly common that such phases occur repeatedly. Even in the data-parallel model non-trivial phases of low-concurrency execution appear in multi-scale method implementations.

Secondly, in typical multi-application or even multi-user environments we cannot expect any single application to have exclusive access to the hardware resources. Consequently, applications compete for resources in an uncontrolled and non-cooperative way as multiple applications start and stop at unpredictable and unforeseeable times. The operating system layer organises the resulting resource oversubscription in a correct but not in an efficient way, as we discuss in the following section.

2 Resource administration vs resource management

The operating system is the canonical layer that administrates computing resources and makes them available to running applications, as illustrated in Fig. 1. However, the operating system does not have any understanding of the internal organisation of concurrent applications. Neither does the operating system know the user's preferences regarding the placement of compute tasks with respect to the hierarchical memory organisation and the resulting opportunities for performance and energy saving.

In an undersubscribed system the operating system could follow essentially one of two policies. Compute tasks could be spread out as much as possible over the system. For instance, four tasks could be run on one core of each processor. As a consequence, each task would benefit from the entire cache memory and the whole memory bandwidth of the socket. Alternatively, the operating system could choose to concentrate all four tasks on a single socket. In this case the

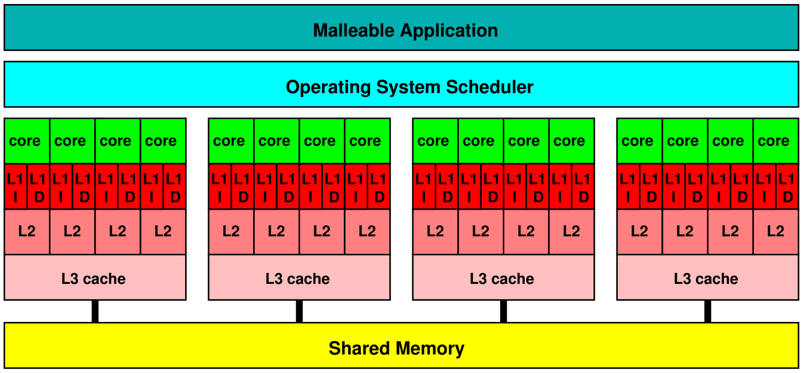


Fig. 1. Architectural model of a cache-coherent shared memory system with four sockets, each equipped with a quad-core processor, and a hierarchical memory organisation with shared L3 caches per processor and individual L2 and L1 (instruction and data) caches for each core

tasks must share the limited cache capacity and memory bandwidth, but on the other hand, the tasks could very efficiently communicate via the shared L3 cache and the other three sockets could be shut down for maximum energy savings in the absence of sufficient useful computing tasks.

In an oversubscribed system the situation changes profoundly. Now, all cores would be busy computing all the time, but in order to ensure fair progress of all tasks, despite the limited computing resources, the operating system resorts to pre-emptive scheduling and time slicing, i.e., an executable task is assigned to a core for execution for a bounded time interval only at whose end it is replaced by another task waiting for execution. This solution stems from the times when uniprocessor systems were mimicking parallel systems where multiple interactive applications were supposed to all make progress and remain responsive. In our scenario of compute-bound applications time-slicing has a rather negative effect on performance.

With multiple compute-bound tasks time-slicing mainly causes overhead for stopping one task, saving its execution state and re-installing another task from the ready queue for execution. In addition to executing the necessary instructions we need to switch from user mode execution to kernel mode execution, which is particularly expensive. Moreover, a task over time is typically scheduled to different cores for execution. This has a detrimental effect on data locality as the task's data may still partially by available in core- or socket-local cache elsewhere, but after a context switch needs to be reloaded into a different part of the memory hierarchy.

To avoid costly over-subscription of resources we need runtime systems that specific to a certain concurrent execution model (e.g. streaming, divide-and-conquer or data-parallel) map the concurrency effectively exposed by an application to a fixed set of worker kernel threads as the common software abstraction of shared memory parallel systems. Fig. 2 illustrates the resulting system architecture. In this model the runtime system cooperates with the operating system such that the runtime system makes the educated decisions while it employs the runtime system to actully implement the descisions.

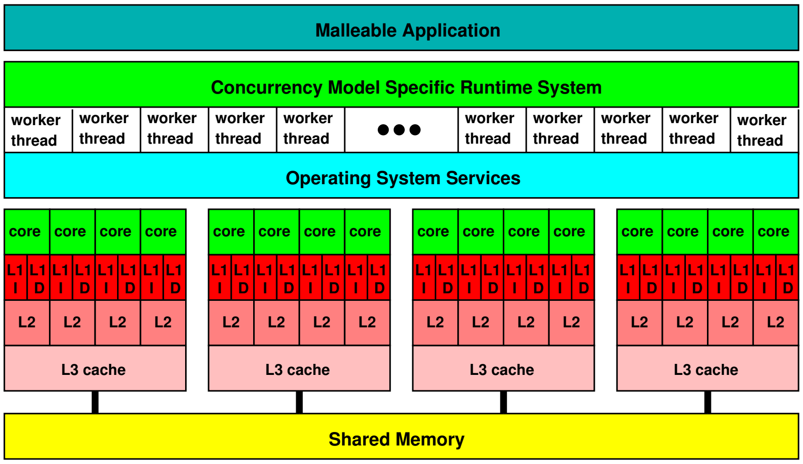


Fig. 2. Architectural model of a cache-coherent shared memory system, as in Fig. 1, with two layers of system software between applications and hardware: operating system and concurrency-model specific runtime system

A *resource management server* dynamically allocates execution resources to a running S-Net program. The (fine-grained) tasks managed by the runtime system are automatically mapped to the dynamically varying number of effectively available kernel threads. Their number is continuously adapted to the effective level of concurrency exposed by the running S-Net streaming network.

In this way, we actively control the energy consumption of a system and reduce the energy footprint of a resource management enabled application compared to greedy resource utilisation, assuming that the underlying operating system automatically reduces the clock frequency and potentially the voltage of underutilised processors and cores or switches them off entirely. Furthermore, we create the means to simultaneously run multiple independent and mutually unaware resource management enabled applications on the same set of resources by continuously negotiating resource distribution proportional to demands.

In contrast to an application-unaware operating system our approach has the advantage that the resource management server understands both sides: the available resources in the computing system **and** the parallel behaviour of the resource management aware running applications. This is why we expect to achieve better performance and less energy consumption compared to today's multi-core operating systems.

3 Managing resource under-subscription

With resource over-subscription effectively solved by a system of worker threads we now come back to a question raised earlier: how many worker threads to actually use in practice. Obviously, using more worker threads than cores leads to resource over-subscription and thus is undesirable. However, as argued earlier any fixed number of worker threads throughout the entire application live time is likely not to be ideal either as soon as applications expose varying levels of exploitable concurrency. If an application at certain times cannot make effective use of all resources, it would be very desirable to either shut down surplus resources for energy saving or, alternatively, make the resources available to other applications.

Active resource management is a runtime system service that dynamically allocates execution resources on demand. A dedicated resource server (thread) is responsible for dynamically spawning and terminating worker threads as well as for binding worker threads to execution resources like processor cores, hyperthreads or hardware thread contexts, depending on the architecture being used.

Upon program startup only the resource server thread is active; this is the master thread of the process. The resource server thread identifies the hardware architecture the process is running on. Next, the resource server sets up the static property graph, which is to be shared by all worker threads. Once the set up is completed, the resource server launches the first worker thread.

Creation (and termination) of worker threads is controlled by the resource service making use of two *resource level indicators*. The first one is the obvious number of currently active worker threads. This is initially zero. The second resource level indicator is a measure of *demand for compute power*. This reflects the number of work queues in the runtime system. The demand indicator is initially set to one. Both resource level indicators are restricted to the range between zero and the total number of hardware execution resources found in the system.

If the demand for computing resources is greater than the number of workers (i.e. the number of currently employed computing resources), the resource server spawns an additional worker thread. Initially, this condition holds trivially. The creation of an additional worker thread temporarily brings the (numerical) demand for resources into an equilibrium with the number of actively used resources. Before increasing the demand the new worker thread must actually find some work to do. Once doing productive work, the worker signals this to the resource server, and the resource server increments the demand level indicator, unless demand (and hence resource use) has already reached the maximum for the given architecture. This procedure guarantees a smooth and efficient organisation of the ramp up phase.

If an application exposes less concurrency work queues of workers may run empty. The worker signals this state to the resource server, which in turn reduces the demand level indicator by one. The worker thread does not immediately terminate because we would like to avoid costly repeated termination and re-creation of worker threads in not uncommon scenarios of oscillating resource demand. The worker thread, however, does effectively terminate with a configurable delay following an extended period of inactivity.

4 Multiple independent applications

The next step in advancing the concept of active resource management is to address multiple independent and mutually unaware applications (or instances thereof) that run at overlapping intervals of time on the same set of execution resources. Fig. 3 illustrates our approach with two applications. We effectively split our resource management service into two parts: a local resource service manages the worker threads within an application, whereas a system-wide resource service is in charge of the computing resources as a whole and effectively mediates these resources between multiple competing applications. This system resource service is started prior to any resource management enabled application process.

Whenever an application has reason to spawn one more worker thread, it first must contact the system resource service to obtain another execution resource. The system resource service either replies with a concrete core identifier or it does not reply at all. In the former case the aplication resource service spawns another worker thread and binds it to the given core. In the latter case the number of execution resources currently occupied by this application remains as is.

Fig. 3 illustrates the simulation of two malleable applications on an 8-core system. For simplicity we ignore any hierarchy in system architecture here. We begin with the start of application 1 on an idle system. Application 1 incrementally allocates all 8 cores via the system resource service. As application 1 apparently exposes sufficient concurrency internally that the application-level resource service actually decides to go this way. At some point application 1 runs concurrently on all 8 cores of the system.

Now we start application 2. Initially, there are no resources whatsoever to run application 2. Thus, application 2 merely requests one core from the system resource service. The system resource service currently has no resources to allocate, but it requests from application 1, more precisely from that application's resource service, to vacate one core. The runtime system of application 1 reacts to this request in an appropriate way and vacates one core at the earliest possible time. Once returned to the system resource service, the latter immediately assigns that core to application 2, which only now effectively begins to run.

Assuming both applications expose ample concurrency, the procedure repeats 3 times until both applications share the 8 cores in a fair way. At times applications run through phases of less concurrency. At some time application 1 deliberately returns a core to the system resource service, which is immediately given to application 2. In a later stage both applications can only make effective use of 3 cores each, and, thus, 2 cores remain empty and could be powered down if the hardware allows.

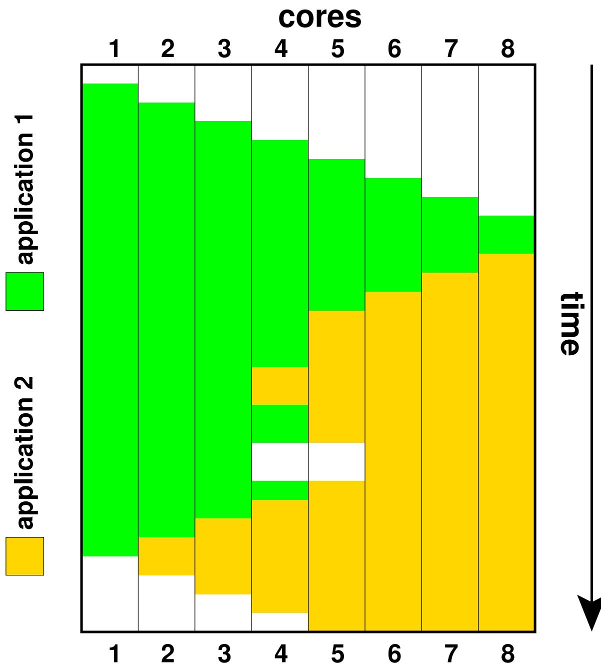


Fig. 3. Simulation of a fictive example with two independent applications

Eventually, application 1 approaches termination and its concurrency fades out. The vacant resources are immediately transferred to application 2 by the system resource service before also application 2 begins to fade out and step-by-step returns cores to the system resource service.

5 Related work

The work closest to our's is the concept of *invasive computing*, advocated by Teich et al [9, 10]. Here, application programs execute a cycle of four steps:

1. explore resources,
2. invade resources,
3. compute,
4. retreat / vacate resources.

Whereas these steps in one way or another can also be found in our proposal, the fundamental difference between their work and our's is the following: Teich et al demand every application to explicitly implement the above steps and provide an API to do so. In contrast, we develop a runtime system that automatically mediates between malleable but otherwise resource-unaware applications and a set of hardware resources that only become known at application start and are typically shared by multiple applications.

Other related work can be found in the general area of operating system process/thread scheduling. Operating systems have long had the ability to map dynamically changing numbers of processes (or kernel threads) to a fixed set of computing resources. However, operating systems do this in an application-agnostic way as they cannot affect the number of processes or threads created. They can merely admister them. As long as the number of processes is less than the number of resources, various mapping policies can be thought of like in our solution. As soon as the number of processes exceeds the number of resources, an operating system resorts to preemptive time slicing.

This all makes sense as long as one takes the resource demands of applications as fixed, but exactly that assumption does not hold for malleable applications. More precisely, malleable applications do have the freedom to adjust resources internally. Trouble is that the application programmer effectively can hardly make use of this opportunity as she or he has no indication of what a good policy could be at application runtime. The operating system, on the other hand, can only react on applications' demands, but not control or affect them in any way. This is exactly where our runtime system support kicks in.

6 Conclusion and future work

We presented active resource management for malleable applications. Instead of running an application on all available resources (or some explicitly defined subset thereof), our runtime system service dynamically adjusts the actually employed resources to the continuously varying demand of the application as well as the continuously varying system-wide demand for resources in the presence of multiple independent applications running on the same system.

Our motivation for this extension is essentially twofold. Firstly, we aim at reducing the energy footprint of streaming applications by shutting down system resources that at times we cannot make effective use of due to limitations in the concurrency exposed. Secondly, we aim at efficiently mediating the available resources among several S-Net streaming applications, that are independent and unaware of each other.

We are currently busy implementing the proposed runtime system techniques within the Front runtime system of S-Net, which is one of many variants of the model runtime system described earlier in the paper. As future work we plan to run extensive experiments demonstrating the positive effect on system-level performance of multiple applications as well as their accumulated energy footprint.

References

1. Dagum, L., Menon, R.: OpenMP: An Industry-Standard API for Shared-Memory Programming. IEEE Transactions on Computational Science and Engineering **5** (1998)

2. Grelck, C., Scholz, S.B.: SAC: A functional array language for efficient multithreaded execution. International Journal of Parallel Programming **34** (2006) 383–427
3. Grelck, C.: Shared memory multiprocessor support for functional array processing in SAC. Journal of Functional Programming **15** (2005) 353–401
4. Ayguade, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Teruel, X., Unnikrishnan, P., Zhang, G.: The Design of OpenMP Tasks. IEEE Transactions on Parallel and Distributed Systems **20** (2009) 404–418
5. Blumofe, R., Joerg, C., Kuszmaul, B., Leiserson, C., Randall, K., Zhou, Y.: Cilk: An Efficient Multithreaded Runtime System. Journal of Parallel and Distributed Computing **37** (1996) 55–69
6. Grelck, C., Scholz, S.B., Shafarenko, A.: A Gentle Introduction to S-Net: Typed Stream Processing and Declarative Coordination of Asynchronous Components. Parallel Processing Letters **18** (2008) 221–237
7. Grelck, C., Scholz, S., Shafarenko, A.: Asynchronous Stream Processing with S-Net. International Journal of Parallel Programming **38** (2010) 38–67
8. Gijsbers, B., Grelck, C.: An efficient scalable runtime system for macro data flow processing using s-net. International Journal of Parallel Programming **42** (2014) 988–1011
9. Teich, J., Henkel, J., Herkersdorf, A., Schmitt-Landsiedel, D., Schröder-Preikschat, W., Snelting, G.: Invasive computing: An overview. In Hübner, M., Becker, J., eds.: Multiprocessor System-on-Chip. Springer (2011) 241–268
10. Teich, J., Weichslgartner, A., Oechslein, B., Schröder-Preikschat, W.: Invasive computing — concepts and overheads. In: Forum on Specification and Design Languages (FDL 2012). Number 217–224, IEEE (2012)

High-Performance Language Composition: Supporting C Extensions for Dynamic Languages

An abbreviated version of [10].

Grimmer Matthias[1], Chris Seaton[2], Thomas Würthinger[2] and Hanspeter Mössenböck[1]

[1] Johannes Kepler University, Linz, Austria
{grimmer,moessenboeck}@ssw.jku.at
[2] Oracle Labs
{chris.seaton,thomas.wuerthinger}@oracle.com

Abstract. Many dynamic languages such as Ruby offer functionality for writing parts of applications in a lower-level language such as C. These C extension modules are usually written against the API of an interpreter, which provides access to the higher-level language's internal data structures. Alternative implementations of the high-level languages often do not support such C extensions because implementing the same API as in the original implementations is complicated and limits performance.
In this paper we describe a novel approach for modular composition of languages that allows dynamic languages to support C extensions through interpretation. We propose a flexible and reusable cross-language mechanism that allows composing multiple language interpreters. This mechanism allows us to efficiently exchange runtime data across different interpreters and also enables the dynamic compiler of the host VM to inline and optimize programs across multiple language boundaries.
We evaluate our approach by composing a Ruby interpreter with a C interpreter. We run existing Ruby C extensions and show how our system executes combined Ruby and C modules on average over 3× faster than the conventional implementation of Ruby with native C extensions.

1 Introduction

Most programming languages offer functionality for calling routines in modules that are written in another language. There are multiple reasons why programmers want to do this, including to run modules already written in another language, to achieve higher performance than is normally possible in the primary language, or generally to allow different parts of the system to be written in the most appropriate language.

Dynamically typed and interpreted languages such as Perl, Python and Ruby often provide support for running extension modules written in the lower-level language C, known as C extensions. C extensions are written in C or a language, which can meet the same ABI such as C++, and are dynamically loaded and linked into the interpreter as a program runs. The APIs that these extensions are written against often simply provide direct access to the internal data structures of the primary implementation of the language. For example, C extensions for Ruby are written against the API of the original implementation of Ruby, known as MRI[3]. This API contains functions that allow C code to manipulate Ruby objects at a high level and to add C implementations of functions.

This model for C extensions worked well for the original implementations of these languages. As the API directly accesses the implementation's internal data structures, the interface is powerful, has low overhead, and was simple for the original implementations to add: all they had to do was make their header files public and support dynamic loading of native modules. However, as popularity of these languages has grown, alternative projects have increasingly attempted to re-implement them using modern virtual machine technology such as dynamic or just-in-time (JIT) compilation or advanced garbage collection. Such projects typically use significantly different internal data structures to achieve better performance, so the question therefore is how to provide the same API that the C extensions expect.

For these reasons, modern implementations of dynamic languages often have limited support for C extensions. For example, the JRuby [17] implementation of Ruby on top of the Java Virtual Machine (JVM) [4] had limited experimental support for C extensions until this was removed after the work proved to be too complicated to maintain and the performance too limited [2,6]. Lack of support for C extensions is often given as one of the major reasons for the slow adoption of modern implementations of such languages.

We would like to enable modern implementations of languages to support C extensions with minimal cost for implementing the existing APIs, and without preventing any advanced optimizations that these implementations use to improve the performance.

Our goal is to run multi-language applications on separate language interpreters, but within the same virtual machine and based on a common framework and using the same kind of intermediate representation. We propose a novel mechanism that allows composing these interpreters, rather than accessing foreign functions and objects via an FFI. Foreign objects and functions are accessed by sending language-independent messages. We resolve these messages at their first execution with language-specific IR snippets that implement efficient accesses to foreign objects and functions. This approach allows composing interpreters at their AST level and makes language boundaries completely transparent to VM performance optimizations.

To evaluate our approach we composed a Ruby interpreter with a C interpreter to support C extensions for Ruby. In our C interpreter, we substitute all invocations to the Ruby API at runtime with language-independent messages that use our cross-language mechanism. Our system is able to run existing unmodified C extensions for Ruby written by companies and used today in pro-

[3] MRI stands for Matz' Ruby Interpreter, after the creator of Ruby, Yukihiro Matsumoto.

duction. Our evaluation shows that it outperforms MRI running the same C extensions compiled to native code by a factor of over 3.

In summary, this paper contributes the following:

- We present a novel language interoperability mechanism that allows programmers to compose interpreters in a modular way. It allows exchanging data between different interpreters without marshaling or conversion.
- We describe how our interoperability mechanism avoids compilation barriers between languages that would normally prevent optimizations across different languages.
- We describe how we use this mechanism to seamlessly compose our Ruby and C interpreters, producing a system that can run existing Ruby C extensions
- We provide an evaluation, which shows that our approach works for real C extensions and runs faster than all existing Ruby engines.

2 System Overview

We base our work on Truffle [26], a framework for building high-performance language implementations in Java. Truffle language implementations are AST interpreters. This means that the input program is represented as an AST, which can be evaluated by performing an execution action on nodes recursively. All nodes of this AST, whatever language they are implementing, extend a common `Node` class.

An important characteristic of a Truffle AST is that it is *self-optimizing* [27]. Nodes or subtrees of a Truffle AST can replace themselves with specialized versions at runtime. For example, Truffle trees self-optimize as a reaction to type feedback, replacing an add operation node that receives two integers with a node that only performs integer addition and so is simpler. The Truffle framework encourages the optimistic specialization of trees where nodes can be replaced with a more specialized node that applies given some assumption about the running program. If an assumption turns out to be wrong as the program continues to run, a specialized tree can undo the optimization and transition to a more generic version that provides the functionality for all required cases. This self-optimization via tree rewriting is a general mechanism of Truffle for dynamically optimizing code at runtime.

When a Truffle AST has arrived at a stable state with no more node replacements occurring, and when execution count of a tree exceeds a predefined threshold, the Truffle framework partially evaluates [26] the trees and uses the Graal compiler [18] to dynamically-compile the AST to highly optimized machine code. Graal is an implementation of a dynamic compiler for the JVM that is written in Java. This allows it to be used as a library by a running Java program, including the Truffle framework.

In this research we have composed two existing languages implemented in Truffle, Ruby and C.

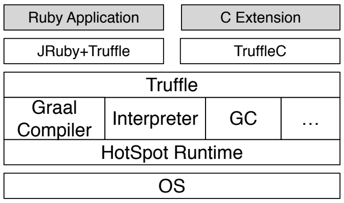


Fig. 1: The layered approach of Truffle: The Truffle framework on top of the Graal VM hosts JRuby+Truffle and TruffleC.

JRuby+Truffle: The Truffle implementation of Ruby [20]. JRuby is the foundation, on which our implementation is built, but beyond the parser and some utilities, little of the two systems are currently shared and JRuby+Truffle should be considered entirely separate from JRuby for this discussion.

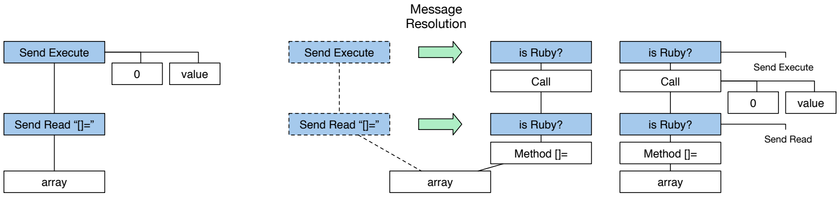
TruffleC: TruffleC [9] is the C language implementation on top of Truffle and can dynamically execute C code on top of a JVM.

Figure 1 summarizes the layered approach of hosting language implementations with Truffle. The Truffle framework provides reusable services for language implementations, such as dynamic compilation, automatic memory management, threads, synchronization primitives and a well-defined memory management. Truffle runs on top of the Graal VM [18,21], a modification of the Oracle HotSpot$^{\mathrm{TM}}$ VM. The Graal VM adds the Graal compiler but reuses all other parts, including the garbage collector, the interpreter, the class loader and so on, from HotSpot.

3 Language Interoperability on Top of Truffle

The goal of our work is to retain the modular way of implementing languages on top of Truffle but make them composable by implementing a cross-language interface. Given this interface, composing two languages, such as C and Ruby, requires very little effort. We do *not* want to introduce a new object model that all Truffle guest languages have to share, which is based on memory layouts and calling conventions. We introduce a common *interface* for objects that is based on code generation via ASTs. Our approach allows sharing language specific objects (with different memory representations and calling conventions) across languages. Finally, we want to make the language boundaries completely transparent to Truffle's dynamic compiler, in that a cross-language call should have exactly the same representation as an intra-language call. This transparency allows the JIT compiler to inline and apply advanced optimizations across language boundaries without modifications.

We use the mechanism to access Ruby objects from C and to forward Ruby API calls from the TruffleC interpreter back to the JRuby+Truffle interpreter.



(a) Using messages to access a Ruby object.

(b) Performing message resolution.

(c) Accessing a Ruby object after message resolution.

Fig. 2: Language independent object access via messages.

Using ASTs as an internal representation of a user program already abstracts away syntactic differences of object accesses and function calls in different languages. However, each language uses its own representation of runtime data such as objects, and therefore the access operations differ. Our research therefore focused on how we can share such objects with different representations across different interpreters.

In this paper we call every non-primitive entity of a program an *object*. This includes Ruby objects, classes, modules and methods, and C immediate values and pointers. An object that is being accessed by a different language than the language of its origin is called a *foreign object*. A Ruby object used by a C extension is therefore considered foreign in that context. If an object is accessed in the language of its origin, we call it a *regular object*. A Ruby object, used by a Ruby program is therefore considered regular. Object accesses are operations that can be performed on objects, e.g. method calls or property accesses.

3.1 Language-independent Object Accesses

In order to make objects (objects that implement `TruffleObject`) *shareable* across languages, we require them to support a common interface. We implement this as a set of *messages*:

Read: We use the *Read* message to read a member of an object denoted by the member's identity. For example, we use the *Read* message to get properties of an object such as a field or a method, and to read elements of an array.

Write: We use the *Write* message to write a member of an object denoted by its identity. Analogous to the *Read* message, we use it to write object properties.

Execute: The *Execute* message, which can have arguments, is used to evaluate an object. For example, it can evaluate a Ruby method or invoke the target of a C function pointer.

Unbox: If the object represents a boxed numeric value and receives an *Unbox* message, this message unwraps the boxed value and returns it. For example, if an *Unbox* message is sent to a Ruby Fixnum, the object returns its value as a 4 byte integer value.

We call an object *shareable* if we can access it via these language-independent messages. Truffle guest-language implementations can insert language-independent message nodes into the AST of a program and send these messages in order to access a foreign object. Figure 2a shows an AST that accesses a Ruby array via messages in order to store `value` at index 0. This interpreter first sends a *Read* message to get the array setter function `[]=` from the array object (in Ruby writing to an element in an array is performed via a method call). Afterwards it sends an *Execute* message to evaluate this setter function. In Figure 2a, the color blue denotes language-independent nodes, such as *message nodes*.

3.2 Message Resolution

The receiver of a cross-language message does *not* return a value that can be further processed. Instead, the receiver returns an AST snippet — a small tree of nodes designed for insertion into a larger tree. This AST snippet contains language-specific nodes for executing the message on the receiver. *Message resolution* replaces the AST node that sent a language-independent message with a language-specific AST snippet that directly accesses the receiver. After message resolution an object is accessed directly by a receiver-specific AST snippet rather than by a message.

During the execution of a program the receiver of an access can change, and so the target language of an object access can change as well. Therefore we need to check the receiver's language before we directly access it. If the foreign receiver object originates from a different language than the one seen so far we access it again via messages and do the message resolution again. If an object access site has varying receivers, originating from different languages, we call the access *language polymorphic*. To avoid a loss in performance, caused by a language polymorphic object access, we embed AST snippets for different receiver languages in an inline cache [12].

Figure 2b illustrates the process of *message resolution* and Figure 2c shows the AST of Figure 2a after message resolution. Message resolution replaced the *Read* message by a Ruby-specific node that accesses the getter function `[]=`. The *Execute* method is replaced by a Ruby-specific node that evaluates this getter method. Message resolution also places other nodes into this AST, which check whether the receiver is really a Ruby object.

Message resolution and building object accesses at runtime has the following benefits:

Language independence: Messages can be sent to any shareable object. The receiver's language of origin does not matter and messages resolve themselves to language-specific operations at runtime.

No performance overhead: Message resolution only affects the application's performance upon the first execution of an object access for a given language. Once a message is resolved and as long as the languages used remain stable, the application runs at full speed.

Cross-language inlining: Message resolution allows the dynamic compiler to inline methods even across language boundaries. By generating AST snippets for accessing foreign objects we avoid the barriers from one language to another that would normally prevent inlining.

3.3 Shared Primitive Values

In order to exchange primitive values across different languages we define a set of *shared primitive types*. We refer to values with such a primitive type as *shared primitives*. The primitive types include signed and unsigned integer types (8, 16, 32 and 64 bit versions) as well as floating point types (32 and 64 bit versions) that follow the IEEE floating point 754 standard.

3.4 JRuby+Truffle: Foreign Object Accesses and Shareable Ruby Objects

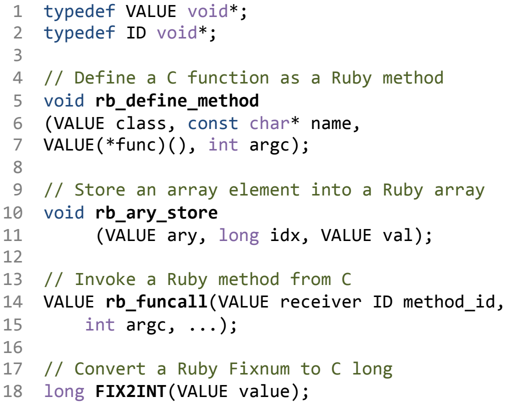
In Ruby's semantics there are no non-reference primitive types and every value is logically represented as an object, as in the tradition of languages such as Smalltalk. Also, in contrast to other languages such as Java, Ruby array elements, hash elements, or object attributes cannot be accessed directly but only via getter and setter calls on the receiver object. For example, a write access to a Ruby array element is performed by calling the `[]=` method of the array and providing the index and the value as arguments.

In our Ruby implementation all runtime data objects as well as all Ruby methods are shareable in the sense that they implement our message-based interface.

Ruby objects that represent numbers, such as `Fixnum` and `Float` that can be simply represented as primitives common to many languages, and also support the *Unbox* message. This message maps the boxed value to the relative shared primitive.

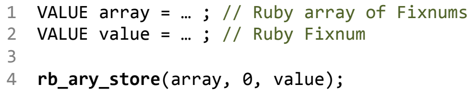
3.5 TruffleC: Foreign Object Accesses and shareable C Pointers

TruffleC can share primitive C values, mapped to *shared primitive values*, as well as pointers to C runtime data with other languages. In our implementation, pointers are objects that implement the message interface, which allows them to be shared across all Truffle guest language implementations. TruffleC represents all pointers (so including pointers to values, arrays, structs or functions) as `CAddress` Java objects that wrap a 64-bit value [11]. This value represents the actual referenced address on the native heap. Besides the address value, a `CAddress` object also stores type information about the referenced object. Depending on the type of the referenced object, `CAddress` objects can resolve the following messages: A pointer to a C struct/array can resolve *Read*/*Write* messages, which access members of the referenced struct/a certain array element. Finally, `CAddress` objects that reference a C function can be executed using the *Execute* message.



```
typedef VALUE void*;
typedef ID void*;

// Define a C function as a Ruby method
void rb_define_method
(VALUE class, const char* name,
VALUE(*func)(), int argc);

// Store an array element into a Ruby array
void rb_ary_store
     (VALUE ary, long idx, VALUE val);

// Invoke a Ruby method from C
VALUE rb_funcall(VALUE receiver ID method_id,
    int argc, ...);

// Convert a Ruby Fixnum to C long
long FIX2INT(VALUE value);
```

Fig. 3: Excerpt of the `ruby.h` implementation.



```
VALUE array = … ; // Ruby array of Fixnums
VALUE value = … ; // Ruby Fixnum

rb_ary_store(array, 0, value);
```

Fig. 4: Calling `rb_ary_store` from C.

TruffleC allows binding foreign objects to pointer variables declared in C. Hence, pointer variables can be bound to `CAdress` objects as well as shared foreign objects.

4 C Extensions for Ruby

Developers of a C extension for Ruby access the API by including the `ruby.h` header file. We want to provide the same API as Ruby does for C extensions, i.e., we want to provide all functions that are available when including `ruby.h`. To do so we created our own source-compatible implementation of `ruby.h`. This file contains the function signatures of all of the Ruby API functions that were required for the modules we evaluated, as described in the next section. We believe it is tractable to continue the implementation of API routines so that the set available is reasonably complete.

Figure 3 shows an excerpt of this header file.

We do not provide an implementation for these functions in C code. Instead, we implement the API by substituting every invocation of one of the functions at runtime with a language-independent message send or directly access the Ruby runtime.

We can distinguish between *local* and *global* functions in the Ruby API:

Local Functions: The Ruby API offers a wide variety of functions that are used to access and manipulate Ruby objects from within C. Consider the function `rb_ary_store` (Figure 4): Instead of a call, TruffleC inserts message nodes into the AST that are sent to the Ruby array (`array`). The AST of the C program (Figure 4) now contains two message nodes (namely a *Read* message to get the array setter method `[]=` and an *Execute* message to eventually execute the setter method, see Figure 2a). Upon first execution these messages are resolved (Figure 2b), which results in a TruffleC AST that embeds a Ruby array access (Figure 2c).

Global Functions: The Ruby API offers various different functions that allow developers to manipulate the global object class of a Ruby application from C or to access the Ruby engine.

The API includes functions to define global variables, modules, or global functions (e.g., `rb_define_method`) etc. In order to substitute invocations of these API functions, TruffleC accesses the global object of the Ruby application using messages or directly accesses the Ruby engine.

Given this implementation of the API we can run C extensions without modification and are therefore compatible with the Ruby MRI API.

5 Evaluation

We evaluated the performance in terms of running time for our implementation of Ruby and C extensions against other existing implementations of Ruby and its C extension API. Ruby is primarily used as a server-side language, so we are interested in peak performance of long running applications after an initial warm-up.

5.1 Benchmarks

We wanted to evaluate our approach on real-world Ruby code and C extensions that have been developed to meet a real business need. Therefore we use the existing modules `chunky_png` [23] and `psd.rb` [19], which are both open source and freely available on the RubyGems website. `chunky_png` is a module that includes routines for resampling, PNG encoding and decoding, color channel manipulation, and image composition. `psd.rb` is a module that includes color space conversion, clipping, layer masking, implementations of Photoshop's color blend modes, and some other utilities.

Both modules have separately available C extension modules to replace key routines with C code, known as `oily_png` [24] and `psd-native` [14], which allows us to compare the C extension against the pure Ruby code. There are 43 routines in the two gems for which a C extension equivalent is provided.

5.2 Compared Implementations

The standard implementation of Ruby is known as **MRI**, or CRuby. It is a bytecode interpreter, with some simple optimizations such as inline caches for method dispatch. MRI has excellent support for C extensions, as the API directly interfaces with the internal data structures of MRI. We evaluated version 2.1.2.

Rubinius is an alternative implementation of Ruby using a significant VM core written in C++ and using LLVM to implement a simple JIT compiler, but much of the Ruby specific functionality in Rubinius is implemented in Ruby. To implement the C extension API, Rubinius has a bridging layer. We evaluated version 2.2.10.

JRuby is an implementation of Ruby on the Java Virtual Machine. JRuby used to have experimental support for running C extensions, but after initial development it became unmaintained and has since been removed. We evaluated the last major version where we found that the code still worked, version 1.6.0.

JRuby+Truffle is our system, using Truffle and Graal. It interfaces to TruffleC to provide support for C extensions. To explore the performance impact of cross-language dynamic inlining, which is only possible in our system, we also evaluated JRuby+Truffle with this optimization disabled.

5.3 Experimental Setup

All experiments were run on a system with 2 Intel Xeon E5345 processors with 4 cores each at 2.33 GHz and 64 GB of RAM, running 64bit Ubuntu Linux 14.04. Where an unmodified Java VM was required, we used the 64bit JDK 1.8.0u5 with default settings. For JRuby+Truffle we used the Graal VM version 0.3. Native versions of Ruby and C extensions were compiled with the system standard GCC 4.8.2.

We ran 100 iterations of each benchmark to allow the different VMs to warm up and reach a steady state so that subsequent iterations are identically and independently distributed. This was verified informally using lag plots [13]. We then sampled the final 20 iterations and took a mean of their runtime as the reported time. We summarize across different benchmarks and report a geometric mean [1].

5.4 Results

Figure 5 shows a summary of our results. We show the geometric mean speedup of each evaluated implementation over all benchmarks, relative to the speed at which MRI ran the Ruby code without the C extension. When using MRI the average speedup of using the C extension (*MRI With C Extension*, Figure 5) over pure Ruby code is around 11×. Rubinius (*Rubinius With C Extensions*, Figure 5) only achieves around one third of this speedup. Although Rubinius generally achieves better performance than MRI for Ruby code [20], its performance for C extensions is limited by having to meet MRI's API, which requires a bridging layer. Rubinius failed to make any progress with 3 of the benchmarks

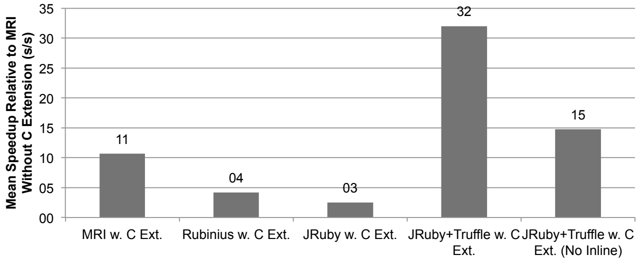


Fig. 5: Summary of speedup across all benchmarks.

in a reasonable time frame so they were considered to have timed out. The performance of JRuby (*JRuby With C Extensions*, Figure 5) is 2.5× faster than MRI running the pure Ruby version of the benchmarks without the C extensions. JRuby uses JNI [15] to access the C extensions from Java, which causes a significant overhead. Hence it can only achieve 25% of the *MRI With C Extension* performance. JRuby failed to run one benchmark with an error about an incomplete feature. As with Rubinius, 17 of the benchmarks did not make progress in reasonable time. Despite a 8GB maximum heap, which is extremely generous for the problems sizes, some benchmarks in JRuby were spending the majority of their time in GC or were running out of heap.

When running the C extension version of the benchmarks on top of our system (*JRuby+Truffle With C Extension*, Figure 5) the performance is over 32× better than MRI without C extensions and over 3× better than *MRI With C Extension*. When compared to the other alternative implementations of C extensions, we are over 8× faster than Rubinius, and over 20× faster than JRuby, the previous attempt to support C extensions for Ruby on the JVM. We also run all the extensions methods correctly, unlike both JRuby and Rubinius.

We can explain this speedup as follows:

In a conventional implementation of C extensions, where the Ruby code runs in a dedicated Ruby VM and the C code is compiled and run natively, the call from one language to another is a barrier that prevents the implementation from performing almost any optimizations. In our system the barrier between C and Ruby is no different to the barrier between one Ruby method and another. We found that allowing inlining between languages is a key optimization, as it permits many other advanced optimizations in the Graal compiler. For example, partial escape analysis [22] can trace objects, allocated in one language but consumed in another, and eventually apply scalar replacement [22] to remove the allocation. Other optimizations that benefit from cross language inlining include constant propagation and folding, global value numbering and strength reduction. When disabling cross-language inlining (*JRuby+Truffle With C Extension (No Inline)*, Figure 5) the speedup over MRI is roughly halved, although it is still around 15× faster, which is around 39% faster than *MRI With C Extension*.

In this configuration the compiler cannot widen its compilation units across the Ruby and C boundaries, which results in performance that is similar to MRI.

If we just consider the contribution of a high performance reimplementation of Ruby and its support for C extensions, then we should compare ourselves against JRuby. In that case our implementation is highly successful at on average over 20× faster. However we also evaluate against MRI directly running native C and find our system to be on average over 3× faster, indicating that our system might be preferable even when it is possible to run the original native code.

6 Related Work

We can compare our work against other projects that seek to compose two languages, and against existing support for C extensions or alternatives in Ruby implementations.

6.1 Unipycation

The work that is closest to our interoperability mechanism is that of Barret et al. [8], in which the authors describe a novel combination of Python and Prolog called Unipycation. We share the same goals, namely to retain the performance of different language parts when composing them and to find an approach that is applicable for any language composition.

However, our approach is quite different both in application and technique. We are concerned in this research in running existing C extensions, so there is immediate utility.

In contrast, Unipycation is a novel combination with no immediate industrial application. Unipycation composes Python and Prolog by combining their interpreters using glue code (which is specific to Python and Prolog) and compiles code using a meta-tracing JIT compiler. In contrast, we do not write glue code for a specific pair of interpreters but rather create this glue code at runtime for any pair of interpreters. Since the IR nodes themselves implement interpretation we can combine IR nodes of different origin without needing glue code.

6.2 Common Language Infrastructure

The Microsoft Common Language Infrastructure (CLI) supports writing language implementations that compile different languages to a common IR and execute it on top of the Common Language Runtime (CLR) [16]. The Common Language Specification (CLS) describes how language implementations can exchange objects across different languages. This standard defines a fixed set of data types and operations that all language implementations have to use. CLS-compliant language implementations generate metadata to describe user-defined types. This metadata contains enough information to enable cross-language operations and foreign object accesses. Also, the CLS specifies a set of basic language features that every implementation has to provide and therefore developers can rely on their availability in a wide variety of languages. This approach is different from ours because it forces CLS-compliant languages to use the same object model. Our approach, on the other hand, allows every language to have its own object model.

6.3 Interface Description Language

Interface Description Languages (IDLs) are also widely used to implement cross-language interoperability. To compose software components, written in different languages, programmers use an IDL to describe the interface of each component. Such IDL interfaces are then compiled to stubs in the host language and in the foreign language. Cross-language communication is done via these stubs [7]. However, an IDL is much more heavyweight. It is mainly targeted to remote procedure calls and often not only aims at bridging different languages but also at calling code on remote computers. Our approach is different because we neither require new interfaces nor a mapping between languages. Foreign objects can be accessed via messages without needing any boilerplate code that converts or marshals an object.

6.4 Language-neutral Object Model

Another approach towards cross-language interoperability are language-neutral object models. Wegiel and Krintz [25] propose a language-neutral object model, which allows different programming languages to exchange runtime data. In their system, the language-neutral objects are stored on an independent shared heap. Each language implementation then transparently translates a shared object to a private object. We argue that sharing objects between different languages and VMs does not require a special object model. Instead, objects should be shared between languages directly. Also, a shared object model would not solve the performance problems that Ruby engines, other than MRI, have when running C extensions.

6.5 Foreign Function Interfaces

Low-level APIs allow developers to integrate C code into another high-level language. Java developers can use a wide variety of different FFIs to integrate C code into Java, for example the Java Native Interface [15], Java Native Access [5], or the Compiled Native Interface [3]. VM engineers that implement new interpreters for dynamic languages in Java, e.g. the original JRuby without Truffle, could use these FFIs to support C extensions. However, the experience of JRuby, described below, shows that this approach is cumbersome and also has limited performance.

Rather than accessing precompiled C extensions via FFIs we follow a completely different approach. We use TruffleC to run these C extensions within a Truffle interpreter and use an efficient cross-language mechanism to compose the JRuby+Truffle and TruffleC interpreter. Our approach hoists optimizations such as cross-language inlining and performs extremely well compared to existing solutions.

6.6 Ruby C Extensions

MRI should have very straightforward support for C extensions as its implementation defines the API. However this does not mean that it poses no problems for MRI. As the interface is well established, MRI is now bound by it as much as any other implementation.

Rubinius supports C extensions through a compatibility layer. This means that in addition to problems that MRI has with meeting a fixed API, Rubinius must also add another layer that converts routines from the MRI API to calls on Rubinius' C++ implementation objects. The mechanism Rubinius uses to optimize Ruby code, an LLVM-based JIT compiler, cannot optimize through the initial native call to the conversion layer.

JRuby uses the JVM's FFI mechanism, JNI, to call C extensions. This technique is almost the same as used in Rubinius, also using a conversion layer, except that now the interface between the VM and the conversion layer is even more complex. In order to exchange data between the JVM and native code, JRuby must copy the data from the JVM onto the native heap. When the native data is then modified, JRuby must copy it back into the JVM. To keep both sides of the divide synchronized, JRuby must keep performing this copy each time the interface is passed.

7 Conclusion

We have presented a new approach to composing implementations of different language interpreters. The cross-language mechanism composes interpreters without additional infrastructure or glue code. We introduce an interface for shareable objects, which allows different language implementations to exchange objects. Language implementations access shared objects via object- and language-independent messages. Our resolving approach transforms these messages to an object- and language-specific access at runtime. The mechanism therefore refrains from converting objects, instead we adapt the IR of a program to deal with the foreign objects. The resolved IR of a program completely obliterates the language boundaries, which enables a JIT compiler to perform its optimizations across any language boundaries.

We use our mechanism to compose the JRuby+Truffle interpreter and the TruffleC interpreter to support C extensions for Ruby. Our evaluation demonstrates that this novel approach exhibits excellent performance. The peak performance of our system is over 3× faster compared to Ruby MRI when running benchmarks which stress interoperability between Ruby code and C extensions.

References

1. How Not To Lie With Statistics: The Correct Way To Summarize Benchmark Results. *Communications of the ACM*, 29, March 1986.
2. Ruby Summer of Code Wrap-Up. `http://blog.bithug.org/2010/11/rsoc`, 2011.
3. CNI (Compiled Native Interface). `http://gcc.gnu.org/onlinedocs/gcj/About-CNI.html`, 2013.
4. HotSpot JVM. Java version history (J2SE 1.3). `http://en.wikipedia.org/wiki/Java_version_history`, 2013.
5. Java Native Access (JNA). `https://github.com/twall/jna#readme`, 2013.
6. jruby-cext: CRuby extension support for JRuby. `https://github.com/jruby/jruby-cext`, 2013.
7. Common Object Request Brooker Architecture (CORBA) Specification. `http://www.omg.org/spec/CORBA/3.3/`, 2014.
8. E. Barrett, C. F. Bolz, and L. Tratt. Unipycation: A case study in cross-language tracing. In *Proceedings of VMIL '13*, New York, NY, USA.
9. M. Grimmer, M. Rigger, R. Schatz, L. Stadler, and H. Mössenböck. TruffleC: Dynamic Execution of C on a Java Virtual Machine. In *Proceedings of PPPJ '14*.
10. M. Grimmer, C. Seaton, T. Wuerthinger, and H. Moessenboeck. Dynamically composing languages in a modular way: Supporting c extensions for dynamic languages. In *Proceedings of MODULARITY '15*.
11. M. Grimmer, T. Würthinger, A. Wöß, and H. Mössenböck. An Efficient Approach for Accessing C Data Structures from JavaScript. In *Proceedings of ICOOOLPS '14*.
12. U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP'91*, pages 21–38.
13. T. Kalibera and R. Jones. Rigorous benchmarking in reasonable time. In *Proceedings of ISMM '13'*.
14. R. LeFevre. PSDNative, 2013.
15. S. Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999.
16. E. Meijer and J. Gough. Technical overview of the common language runtime. *language*, 29:7, 2001.
17. C. Nutter, T. Enebo, O. Bini, N. Sieger, et al. JRuby. `http://jruby.org/`, 2014.
18. Oracle. OpenJDK: Graal project. `http://openjdk.java.net/projects/graal/`, 2013.
19. K. S. Ryan LeFevre et al. PSD.rb from Layer Vault, 2013.
20. C. Seaton, M. L. Van De Vanter, and M. Haupt. Debugging at full speed. In *Proceedings of DYLA'14*. ACM.
21. L. Stadler, G. Duboscq, H. Mössenböck, and T. Würthinger. Compilation queuing and graph caching for dynamic compilers. In *Proceedings of VMIL '12*.
22. L. Stadler, T. Würthinger, and H. Mössenböck. Partial escape analysis and scalar replacement for java. In *Proceedings of CGO '14*.
23. W. van Bergen et al. Chunky PNG, 2013.
24. W. van Bergen et al. OilyPNG, 2013.
25. M. Wegiel and C. Krintz. Cross-language, type-safe, and transparent object sharing for co-located managed runtimes. Technical Report 2010-11, UC Santa Barbara, 2010.
26. T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to rule them all. In *Proceedings of ONWARD 2013*. ACM.
27. T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. Self-optimizing AST interpreters. In *Proceedings of DLS'12*.

Programming GPUs with C++14 and Just-In-Time Compilation

Michael Haidl, Bastian Hagedorn, and Sergei Gorlatch

University of Muenster
{m.haidl, b.hagedorn, gorlatch}@uni-muenster.de

Abstract. Systems that comprise accelerators (e.g., GPUs) promise high performance, but their programming is still a challenge, mainly because of two reasons: 1) two distinct programming models have to be used within an application: one for the host CPU (e.g., C++), and one for the accelerator (e.g., OpenCL or CUDA); 2) using Just-In-Time (JIT) compilation and its optimization opportunities in both OpenCL and CUDA requires a cumbersome preparation of the source code. These two aspects currently lead to long, poorly structured, and error-prone GPU codes. Our PACXX programming approach addresses both aspects: 1) parallel programs are written using exclusively the C++ programming language, with modern C++14 features including variadic templates, generic lambda expressions, as well as STL containers and algorithms; 2) a simple yet powerful API (PACXX-Reflect) is offered for enabling JIT in GPU kernels; it uses lightweight runtime reflection to modify the kernel's behaviour during runtime. We show that PACXX codes using the PACXX-Reflect are about 60% shorter than their OpenCL and CUDA Toolkit equivalents and outperform them by 5% on average.

1 Introduction

Accelerators such as Graphics Processing Units (GPUs) are increasingly used in today's high-performance systems. However, programming such systems remains complicated, because it requires the use of two distinct programming models: one for the host CPU (e.g., C or C++) and one for the GPU (e.g., OpenCL or CUDA). The codes for GPUs (so-called kernels) are written using limited subsets of the C/C++ language which miss many advanced features of the current standards like C++14 [1]. Furthermore, specific language constructs of CUDA and OpenCL for parallelization and synchronization have to be additionally mastered by the GPU software developers. Last but not least, the Just-In-Time (JIT) compilation which is a proven technique of simplifying and optimizing the programming process is provided for GPUs on a very restricted scale.

This paper aims at simplifying and improving the programming process for systems with GPUs and other accelerators by making two main contributions:

1. We present and implement *PACXX (Programming Accelerators with C++)* - a unified programming model based on the newest C++14 standard that uniformly covers both host and kernel programming without any language extensions.

2. We develop *PACXX-Reflect* – a simple yet powerful API to enable lightweight JIT compilation of PACXX programs in order to optimize the kernel code during program execution by using values that become known at runtime.

We evaluate our programming approach for GPUs with C++14 and JIT compilation using two case studies – matrix multiplication and Black-Scholes model for the option market – and demonstrate that PACXX codes are about 60% shorter than their manually optimized OpenCL and CUDA Toolkit equivalents and outperform them 5% on average.

The structure of the paper is as follows. Section 2 provides an overview of the state of the art and related work in programming systems with accelerators in general and the JIT compilation approaches for such systems in particular. In Section 3, we explain the PACXX programming approach and compare it to CUDA by way of example. We present our approach to JIT compilation using the PACXX-Reflect API in Section 4.

In Section 5, we briefly describe the implementation of PACXX and we evaluate our approach on two case studies by comparing the size and performance of PACXX codes to the corresponding CUDA and OpenCL programs. Finally, we conclude in Section 6.

2 State of the Art and Related Work

In current programming approaches like CUDA and OpenCL, the code for a GPU (so-called kernel) is written using a limited subset of the C/C++ language, e.g., CUDA C++ [3] which misses many advanced features of the current standards like C++14 [1]. Memory management for the GPU memory has to be performed by the developer explicitly, because C++ has no language support for distinct memories. Memory is explicitly allocated twice - first in the host memory and then again in the GPU's memory. The developer is also responsible for performing explicit synchronization (copying) between the two distinct memories. This implies a significantly longer boilerplate (host-)code for memory management as compared to C++ where allocation and initialization are performed together through the RAII (Resource Acquisition Is Initialization) idiom [4].

Although CUDA and OpenCL (in a provisional version of the OpenCL 2.1 standard [5]) have been recently extended with static C++11 language features, these C++ extensions define new, for C++ developer unfamiliar, language elements (e.g., cast operators), while dynamic language features of C++ such as the Standard Template Library are still not provided by neither CUDA nor OpenCL.

Several recent approaches aim at integrating the accelerator programming into C++. The C++ AMP approach [6] extends C++ by an explicit data-parallel language construct (`parallel_for_each`), and so-called `array_views` provide functions for memory transfers. The developer still needs to use a wrapper (i.e., write an additional line of code) for each memory allocation and use the C++ AMP views instead of the original C++ data types in order to achieve that memory synchronization is done transparently by the system. SYCL [7] is a high-level interface that integrates the OpenCL programming model into C++ by using the lambda features of the C++11 standard, but it still demands multiple memory allocations for so-called `Buffers` both in the host and kernel code. Nvidia Thrust [8] and AMD Bolt [9] are libraries implementing the functionality of the C++ Standard Template Library (STL) in a data-parallel way, but they are restricted to accelerators from the corresponding vendor and do not support modern C++14 language features. Annotation-based approaches OpenMP [10] and OpenACC [11] expect the user to use parallelism-specifying directives in addition to C++. STAPL [12] offers STL functionality which is executed in parallel by the underlying runtime system, but it targets distributed-memory systems rather than systems with GPUs.

None of the described programming models offers the possibility to transform and optimize the kernel code during execution, i.e., in a Just-In-Time (JIT) manner. For example, writing a kernel for a particular size of input data provides usually better performance than a generic kernel, due to additional optimizations performed by the compiler. However, writing separate kernels for different data sizes would lead to a poorly structured, hardly maintainable codes. Just-in-time compilation can be used to optimize code by taking into account values which become known during the execution of the program: thereby, compilers can additionally optimize code when performance-critical variables (e.g., exit conditions of loops) are resolved.

OpenCL and the newest releases of CUDA support JIT compilation of kernel code. However, both approaches demand that the kernel code is provided as human-readable code which has a security drawback: the source code may be disclosed to non-authorized parties. The NVRTC [13] library supports all C++ features available in CUDA for JIT compilation. Unfortunately, a complicated problem arises using NVRTC: to allow function overloading and template functions as kernels, CUDA C++ follows the C++ standard regarding function name mangling while kernel names are machine generated and unknown to the developer. Without knowing the mangled name of a kernel, the function pointer for invoking the kernel cannot be retrieved and the kernel cannot be called. Additionally, template kernels must be explicitly instantiated prior to their use by the developer. The current solution is to enforce the C naming policy using `extern "C"`, but this completely disables function overloading and templates for kernels, because function names are no longer unique and the right kernel cannot be resolved by its name. Another solution could be an additional library providing function name de-mangling, but this would introduce more development overhead and unnecessary boilerplate code, because function names would have to be combined with the actual data types used in the templates. Another recent JIT approach is LambdaJIT [14] that automatically parallelizes the lambda functions used in STL algorithms. Through different back-ends, LambdaJIT is capable of offloading computations to a GPU.

This paper describes a novel approach to programming systems with GPUs and other accelerators: our PACXX model relies exclusively on the newest C++14 standard without any extensions, and we use lightweight runtime reflection to provide JIT compilation and optimization of PACXX programs. PACXX provides the programmer with all advanced features of C++14, e.g., variadic templates, generic lambda expressions, as well as STL containers and algorithms. Reflection enables a program to modify its behaviour and is well known in languages like Java and Scala [15] but is unavailable in C++. In contrast to other approaches like Reflex [16] and XCppRefl [17] which aim to integrate full runtime reflection of the type system into C++, PACXX-Reflect follows a lightweight approach: we do not perform any type analysis or introspection.

3 The PACXX Programming Model

To explain our C++-based approach to GPU programming, we consider a popular example in parallel computing: vector addition. We start with a C++ program and then compare how this program is parallelized for GPU using CUDA vs. our PACXX approach.

```
1  int main(){
2    vector<int> a{N}, b{N}, c{N};
3    for (int i = 0; i < N; ++i)
4      c[i] = a[i] + b[i];
5  }
```

Listing 1.1: Sequential vector addition in C++.

Listing 1.1 shows the sequential C++ source code for the vector addition. The memory is allocated and initialized with the default constructor of the vector's element type, here with 0 – following the RAII (Resource Acquisition Is Initialization) idiom – during the construction of the three STL containers of type `std::vector` in line 2. The calculation is performed by the for-loop in line 3.

```
1  __global__ void vadd (int* a,
2      int* b, int* c, size_t size){
3    auto i = threadIdx.x
4         + blockIdx.x * blockDim.x;
5    if (i < size)
6        c[i] = a[i] + b{i];
7  }
8  int main(){
9    vector<int> a{N}, b{N}, c{N};
10   int* da, db, dc;
11   cudaMalloc(da, N*sizeof(int));
12   cudaMemcpy(da,&a[0],sizeof(int)
13     * N, cudaMemcpyHostToDevice);
14   ⋮ // 5 additional lines
15     // for vectors b and c
16   vadd<<<N/1024 + 1, 1024>>>
17               (da, db, dc, N);
18   cudaDeviceSynchronice();
19 }
```

```
int main(){
  vector<int> a{N}, b{N}, c{n};
  auto vadd = kernel([](
      const auto& a, const auto& b
      auto& c){
     auto i = Thread::get().global;
     if (i < a.size())
       c[i.x] = a[i.x] + b[i.x];
  }, {{N/1024 + 1}, {1024}});
  auto F = async(launch::kernel,
              vadd, a, b);
  F.wait();
}
```

Listing 1.2: Two versions of parallel vector addition: in CUDA (left) and PACXX (right).

Listing 1.2 (left) shows how the vector addition is parallelized using CUDA. The CUDA kernel replaces the sequential for-loop of the C++ version with an implicitly data-parallel version of the vector addition. The `vadd` kernel is annotated with the `global` keyword (line 1) and is, according to the CUDA standard, a free function with `void` as return type. In CUDA, functions called by a kernel have to be annotated with the `device` keyword to explicitly mark them as GPU functions. This restriction prohibits the use of the C++ Standard Template Library (STL) in the kernel code because the functions provided in the STL are not annotated and, therefore, callable only from the host code. The parameters of the `vadd` kernel are raw integer pointers. Memory accessed by the kernel must be managed by the developer explicitly (line 11). To use the kernel, memory is allocated on the GPU and the data is synchronized between both memories (line 12). On the host side, three instances of `std::vector` are used for the computation. Since STL features cannot be used in CUDA, three raw integer pointers are defined in line 10 to represent the memory of the GPU. Memory is allocated using the CUDA Runtime API; for brevity, the calls to this API are only shown for one vector. For each allocation and synchronization, an additional line of code is necessary (e.g., line 11). Passing arguments to a kernel by reference which is common in C++, is not possible in CUDA, so we have to use the pointers defined in line 10 and pass them by value. To launch the kernel, a launch configuration must be specified within `<<< >>>` in each kernel call (line 16), i.e., a CUDA-specific, non-C++ syntax is used. CUDA threads are organized in a grid of blocks with up to three dimensions. In our example, 1024 threads are the maximal number of threads in one block, and $N/1024+1$ blocks (N is the size of the vectors) form the so-called launch grid. While all threads execute the same kernel code, the work is partitioned using the index ranges; in our example, each thread computes a single addition depending on its absolute index in the `x`-dimension (line 3). The identifier of the thread within the grid and the block are obtained using variables `threadIdx`, `blockIdx`, `blockDim` and `gridDim` (not shown in the code). To prevent an out-of-bounds access of threads with $i >= N$, a so-called *if guard* (line 5) is used. However, the size of the vector has to be known in the kernel code and is passed as additional parameter to the kernel (line 2). The GPU works asynchronously to the host, therefore, the host execution must be explicitly synchronized with the GPU using the CUDA Runtime API (line 18).

Summarizing, the CUDA code is very different from the original C++ version. A complete restructuring and new language constructes are necessary, and the size of code increases significantly.

Listing 1.2 (right) shows the PACXX source code that performs the same computation. It is a pure C++14 code without any extensions (e.g., new keywords or special syntax), with the kernel code inside of the host code, that uses `std::async` and `std::future` from the STL concurrency library [1] to express parallelism on the GPU. In PACXX, there are no restrictions which functions can be called from kernel code, however, the code must be available at runtime, i.e., functions from pre-compiled libraries cannot be called. As in CUDA, a PACXX kernel is implicitly data parallel: it is defined with a C++ lambda function (lines 5-9). PACXX provides the C++ template class `kernel` (line 4) to identify kernels: instances of `kernel` will be executed in parallel on the GPU. The launch configuration (the second parameter in line 9) is defined analogous to CUDA. As in CUDA, threads are identified with up to three-dimensional index ranges which are used to partition the work amongst the GPU's threads. The thread's index can be obtained through the `Thread` class (line 6). To retrieve values from the thread's block, the class `Block` is available. The kernel instance created in line 3 is passed to the STL-function `std::async` (line 10) that invokes the kernel's execution on the GPU. PACXX provides an STL implementation based on libc++ [18] where an additional launch policy (`launch::kernel`) for `std::async` is defined to identify kernel launches besides the standard policies of the C++ standard (`launch::async` and `launch::deferred`). Passing the `launch::kernel` policy to `std::async` (line 10) implies that the kernel should be executed on the GPU, rather than by another host thread. The additional parameters passed to `std::async` (line 11) are forwarded to the kernel. As a significant advantage over CUDA, parameters of a kernel in PACXX can be passed by reference, as with any other C++ function. PACXX manages the GPU memory implicitly [2], i.e., no additional API for memory management (as in CUDA and OpenCL) has to be used by the developer. Non-array parameters passed by reference are copied to the GPU prior to the kernel launch and are automatically synchronized back to the host when the kernel has finished. For arrays, the `std::vector` and `std::array` classes are used. As in the original C++ version, memory is allocated using the `std::vector` class (line 2). PACXX extends the STL containers `std::vector` and `std::array` with the lazy copying strategy: synchronization of the two distinct memories happens only when a data access really occurs, either on the GPU or on the host. This happens transparently for the developer and reduces the programming effort significantly. As compared, to the CUDA version, the PACXX code only needs 13 LoC, i.e., it is almost 50% shorter. The PACXX kernel also requires an *if guard* to prevent out-of-bounds access, but there is no need to pass the size of the vectors to the kernel as an additional parameter, because each instance of `std::vector` knows the number of contained elements which can be obtained by the vector's `size` function (line 7). The `std::async` function used to execute the kernel on the GPU returns an `std::future` object associated with the kernel launch (line 10); this object is used to synchronize the host execution with the asynchronous kernel more flexibly than in CUDA: the `wait` member function (line 12) blocks the host execution if the associated kernel has not finished yet.

Due to the exclusive usage of the C++14 syntax and implicit memory management of kernel parameters and STL containers, GPU programming using PACXX is shorter and easier for C++ developers than when using CUDA.

4 The PACXX-Reflect API

To exploit JIT compilation in CUDA or OpenCL, the source code must be prepared manually by the developer; it is commonly represented as a string, either in the executable itself or loaded from a source file.

Listing 1.3 compares the JIT-compilable version of vector addition in CUDA (left) and PACXX (right). In CUDA, a placeholder (`"#VSIZE#"`) is introduced into the kernel code. During program execution this placeholder is replaced by the string with the actual size of vector `a` (line 12– 19). However, changing the kernel code at runtime requires additional programming effort in the host code and, therefore, increases the program length. Furthermore, during the JIT compilation the kernel code passes all compiler stages including parsing, lexing and the generation of an abstract syntax tree (AST). Since the AST for the kernel code is created at runtime of the program, errors in the kernel code are found first at program execution which makes debugging more

```
1  vector<int> a{N}, b{N}, c{N};     vector<int> a{N}, b{N}, c{N};
2  string str{ R"(
3  extern "C"                        auto vadd = [](const auto& a,
4  __global__ void vadd(int* a,                     const auto& b,
5                  int* b, int* c){                 auto& c){
6    auto i = threadIdx.x              auto i = Thread::get().global.x;
7         + blockIdx.x * blockDim.x;   auto size =
8    int size = #VSIZE#;               reflect([&]{return a.size();});
9    if (i < size)                     if (i < size)
10     c[i] = a[i] + b[i];               c[i] = a[i] + b[i];
11 })" };                            };
12 string v{to_string(a.size())};
13 string p{"#VSIZE#"};
14 string vadd{str};
15 auto npos = string::npos;
16 for(size_t i = 0;
17   (i = vadd.find(p, i))!= npos;
18    i += v.size())
19   vadd.replace(i, p.size(), v);
```

Listing 1.3: JIT-compilable vector addition in CUDA (left) and PACXX (right).

difficult. With the vector's size hard-coded into the kernel code, the additional parameter for the vector size as in Listing 1.2 becomes unnecessary, but this is only a minor optimization since kernel parameters are stored in the very fast constant memory of the GPU.

On the right-hand side of Listing 1.3, the same modification to the *if guard* is made using PACXX-Reflect: the dynamic part of the *if guard* condition is replaced by a constant value using the Reflect API. This is accomplished using the `reflect` function which is defined as a variadic template function that takes a lambda expression and an arbitrary number of additional parameters. The `reflect` function forwards the return value of the lambda expression passed to it (line 8); the expression is evaluated in the host's context, i.e., the instances of the `reflect` template function are JIT-compiled for the host architecture and executed on the host prior to the kernel's launch. Only constant values and kernel parameters are allowed as arguments or captured values by the lambda expression, because they are, from the kernel's point of view, compile-time constant values. In Listing 1.3 (right), a direct call to the `size` function of the `std::vector` requires two loads (of the `begin` and the `end` iterator) from the GPUs memory which might introduce overhead on some architectures. To avoid this, a lambda expression which wraps the call to the `size` function of vector `a` is passed to the `reflect` function; the returned value from `reflect` is considered static for the kernel execution and the function call is replaced by the computed value, such that no loads are necessary to retrieve the vector's size. Summarizing, the replacement of the function call with the constant value happens transparently for the developer.

5 Implementation and Evaluation

PACXX is implemented using LLVM [19] and uses its code generation libraries to generate PTX code [20] for Nvidia GPUs and SPIR code [21] for other architectures (e.g., Intel Xeon Phi accelerators). PACXX-Reflect operates on the LLVM IR (intermediate representation) generated by the PACXX offline compiler (implemented in Clang [22]), rather than on the source code itself, thus reducing the overhead of JIT compilation as compared to the current solutions for OpenCL and CUDA and avoids the name mangling problem by design.

For evaluation, two case studies programmed in C++14 and implemented using PACXX are compared to the reference implementations from the CUDA Toolkit [23]: 1) Black-Scholes computation [24] used in high-frequency stock trading, and 2) matrix multiplication [25].

To evaluate the performance of PACXX codes on other architectures than GPUs, the programs from the CUDA Toolkit were also manually re-written in OpenCL. The PACXX and CUDA implementations are compiled at runtime for CUDA Compute Capability 3.5 with standard optimizations enabled (no additional floating point optimizations), using the latest CUDA Toolkit (release 7.0). The CUDA PTX code is generated by the Nvidia NVRTC library; the host code is compiled by Clang 3.6 with standard `O3` optimizations.

For evaluation we use the state-of-the-art accelerator hardware: an Nvidia Tesla K20c GPU controlled by an Intel Xeon E5-1620 v2 at 3.7 GHz, and an Intel Xeon Phi 5110p hosted in a dual-socket HPC-node equipped with two Intel Xeon E5-2695 v2 CPUs at 2.4 GHz. For the Intel Xeon Phi and Intel Xeon, we used the Intel OpenCL SDK 2014. We employed the Nvidia profiler (nvprof) and the OpenCL internal facilities for performance measurements.

5.1 Black-Scholes computation

Our first example is the Black-Scholes (BS) model which describes the financial option market and its price evolution. Both programs, from CUDA Toolkit and the PACXX implementation, compute the Black-Scholes equation on randomly generated input data seeded with the same value to achieve reproducible and comparable results. Input data are generated for $81.92 \cdot 10^6$ options.

Figure 1 shows the measured runtime and the program size in Lines of Code (LoC); the latter is calculated for the source codes formatted by the Clang-format tool using the LLVM coding style. We observe that the OpenCL version (273 LoC) is about 3 times longer and the CUDA version (217 LoC) about 2 times longer than the PACXX code in pure C++ (107 LoC). The advantages of PACXX regarding the code size arise from the tasks performed transparently by the PACXX runtime: device initialization, memory management and JIT compilation, whereas in CUDA and OpenCL these tasks have to be performed explicitly.

The PACXX code has also advantages regarding runtime on the Nvidia K20c GPU (about 4.3%) and on the dual socket cluster node (about 6.7%). On the Intel Xeon Phi, the PACXX code is only about 0.1% slower than the OpenCL version. On the Nvidia Tesla K20c, since the OpenCL implementation from Nvidia is still for the OpenCL 1.1

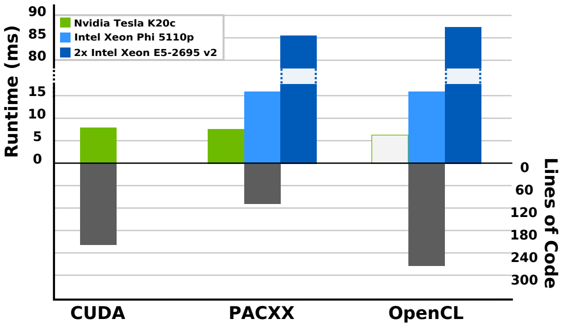


Fig. 1: Evaluation results for the Black-Scholes application.

standard where floating point division are not IEEE 754 compliant, the higher speed of the OpenCL version (16.1% faster than PACXX) is achieved for the lower accuracy.

5.2 Matrix Multiplication

Our second case study is the multiplication of dense, square matrices. The PACXX code uses the Reflect API to compute the sizes of the matrices: the `reflect` function retrieves the input vector's size and computes its square root to get the number of rows/-columns in the matrix. The `reflect` call is evaluated during the runtime compilation and the value (our matrices are of size 4096×4096) is automatically embedded into the kernel's intermediate representation. For the OpenCL and CUDA codes, the values are introduced into the kernel code by string replacement before it is JIT compiled.

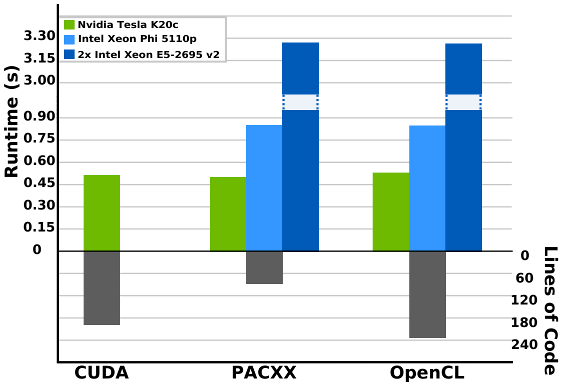


Fig. 2: Evaluation results for matrix multiplication.

Figure 2 (left) shows the runtime and size of the PACXX code as compared to the CUDA code (compiled with NVRTC) and the OpenCL code on the three evaluation platforms. The PACXX program is again much shorter than its CUDA and OpenCL counterparts. We observe that the kernel becomes 2.7% faster on the K20c GPU when using PACXX-Reflect and its JIT capabilities. The PACXX code is about 0.2% slower on the Intel architectures as compared to the OpenCL implementation. On the Nvidia Tesla K20c, PACXX code outperforms CUDA NVRTC and OpenCL codes by 2.6% and 3.3%, correspondingly.

6 Conclusion

We presented PACXX – a programming model for GPU-based systems using C++14 and JIT compilation. We demonstrated that on modern accelerators (GPUs and Intel Xeon Phi) PACXX provides competitive performance and reduces the programming effort by more than 60% of LoCs as compared to CUDA or OpenCL. The code size reduction is achieved through JIT compilation and memory management tasks performed by PACXX implicitly in contrast to CUDA and OpenCL where they must be programmed explicitly. Additionally, PACXX enables application developers to program OpenCL and CUDA capable accelerators (e.g., Nvidia GPU and Intel Xeon Phi) in a modern object-oriented way using all advanced features of C++14 and the STL.

Bibliography

[1] isocpp.org. *Programming Languages - C++ (Committee Draft)*, 2014.

[2] Michael Haidl and Sergei Gorlatch. PACXX: Towards a Unified Programming Model for Programming Accelerators Using C++14. In *Proceedings of LLVM Compiler Infrastructure in HPC (LLVM-HPC) at Supercomputing 14*, pages 1–11. IEEE, 2014.

[3] Nvidia. *CUDA C Programming Guide*, 2015. Version 7.0.

[4] Herb Sutter and Andrei Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. Pearson Education, 2004.

[5] Khronos OpenCL Working Group. *The OpenCL C++ Specification*, 2015. Version 1.0.

[6] Microsoft. *C++ AMP : Language and Programming Model*, 2012. Version 1.0.

[7] Khronos OpenCL Working Group. *SYCL Specifcation*, 2015. Version 1.2.

[8] Nathan Bell and Jared Hoberock. Thrust: A Parallel Template Library. *GPU Computing Gems Jade Edition*, page 359, 2011.

[9] AMD. *Bolt C++ Template Library*, 2014. Version 1.2.

[10] James C. Beyer et al. OpenMP for Accelerators. In *OpenMP in the Petascale Era*, pages 108–121. Springer, 2011.

[11] openacc-standard.org. *The OpenACC Application Programming Interface*, 2013. Version 2.0a.

[12] Ping An et al. STAPL: An Adaptive, Generic Parallel C++ Library. In *Languages and Compilers for Parallel Computing*, pages 193–208. Springer, 2003.

[13] Nvidia. *NVRTC - CUDA Runtime Compilation - User Guide*, 2015.

[14] Thibaut Lutz and Vinod Grover. LambdaJIT: A Dynamic Compiler for Heterogeneous Optimizations of STL Algorithms. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-Performance Computing*, pages 99–108. ACM, 2014.

[15] Martin Odersky et al. The Scala Programming Language. *URL http://www.scala-lang.org*, 2008.

[16] S. Roiser. Reflex - Reflection in C++. In *Proceedings of Computing in High Energy and Nuclear Physics*, 2006.

[17] Tharaka Devadithya, Kenneth Chiu, and Wei Lu. C++ Reflection for High Performance Problem Solving Environments. In *Proceedings of the 2007 Spring Simulation Multiconference*, volume 2, pages 435–440. International Society for Computer Simulation, 2007.

[18] The LLVM Compiler Infrastructure. *libc++ C++ Standard Library*, 2014.

[19] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of International Symposium on Code Generation and Optimization, 2004. CGO 2004*, pages 75–86. IEEE, 2004.

[20] Nvidia. *PTX:Parallel Thread Execution ISA*, 2010. Version 4.2.

[21] Khronos OpenCL Working Group. *The SPIR Specification*, 2014. Version 1.2.

[22] Chris Lattner. LLVM and Clang: Next Generation Compiler Technology. In *Proceedings of the BSD Conference*, pages 1–2, 2008.

[23] Nvidia. *CUDA Toolkit 7.0*, 2015.

[24] Victor Podlozhnyuk. Black-Scholes Option Pricing. *CUDA Toolkit Documentation*, 2007.

[25] Vasily Volkov and James W Demmel. Benchmarking GPUs to Tune Dense Linear Algebra. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis, SC'08.*, pages 1–11. IEEE, 2008.

Approximating Context-Sensitive Program Information

Mathias Hedenborg[1], Jonas Lundberg[1], Welf Löwe[1], and Martin Trapp[2]

[1] Linnaeus University, Software Technology Group,
{Mathias.Hedenborg|Jonas.Lundberg|Welf.Lowe}@lnu.se
[2] Senacor Technologies AG, Martin.Trapp@senacor.com

Abstract. Static program analysis is in general more precise if it is sensitive to execution contexts (execution paths). In this paper we propose χ-terms as a mean to capture and manipulate context-sensitive program information in a data-flow analysis. We introduce *finite k-approximation* and *loop approximation* that limit the size of the context-sensitive information. These approximated χ-terms form a lattice with a finite depth, thus guaranteeing every data-flow analysis to reach a fixed point.

1 Introduction

Static program analysis is an important part of optimizing compilers and software engineering tools. These analyses predict properties of any execution of a given program, referred to as *program information*, by abstracting from its concrete execution semantics and its potential input values. Analyses can be context-sensitive or -insensitive, i.e., an analysis may or may not distinguish program information for different execution paths, i.e. for different *contexts*, e.g., the call contexts of a method. Context-sensitive analyses are, in general, more precise than their context-insensitive counterparts but also more expensive in terms of time and memory consumption.

In an iterative program, there are countably (infinitely) many contexts. Hence, merging the program information of some contexts is needed for the analysis to terminate. This, however, makes the analysis less context-sensitive, hence, less precise.

In Trapp et al. [THLL15], we focussed on capturing context-sensitive analysis information, i.e. contexts and program information for each program point, in a memory efficient way. In other words, we strived to delay merging the program information of different context for keeping analysis precision high. In the present paper, we discuss how to handle the approximations that sacrifies precision for memory. We propose two solutions: *finite k-approximation* and *loop approximation*, and we prove that any context-insensitive data-flow analysis problem have a *k-approximated* context-sensitive counterpart that is guaranteed to reach a fixed point.

The remainder of the paper is structured as follows: Section 2 summarise the notions of χ-terms and some of its fundamental operations. Sections 3 and 4 presents some theoretical results that will be useful later on. Section 5 presents two types of χ-term approximations and discusses how they relate to loop handling and analysis termination. Section 5.5 presents an approximative approach that is a normalized combination of these two χ-term approximations. Section 6 discusses related work, and section 7 concludes the paper and discusses future work.

2 Background

This section summarise the notions of χ-terms and some of its fundamental operations. It is basically a brief summary of [THLL15] included for the completness and understandability of this paper.

2.1 SSA Representation

We assume the analysis to be based on a *Static Single Assignment* (SSA) graph representation [CFR^{+}91] of a program. Nodes in the SSA graph represent program points; special ϕ-nodes represent merge points of the execution paths, i.e., contexts. Here we distinguish the program information of incoming paths by creating a χ-term connected to sub-terms, each representing the program information analyzed for the respective incoming execution path.

Figure 1 shows an example code with corresponding basic block and SSA-graph representations.

```
if (...)
  x = 1;
else
  x = 2
if (...){
  y = x;
  b = 3;
}
else {
  y = 2;
  b = 4;
}
if (...)
  a = x;
else
  a = y;
return a+b;
```

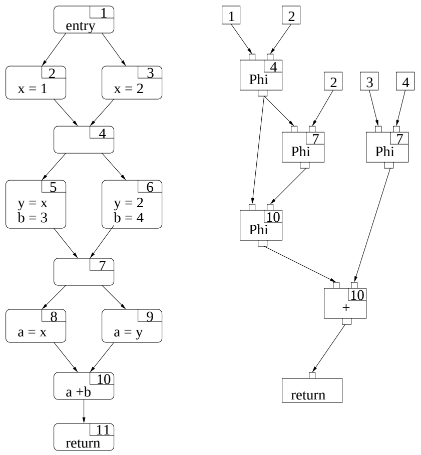


Fig. 1. A source code example with corresponding basic block and SSA graph structures.

In the figure the source code is transfered into numbered basic blocks (middle), and based on this a ϕ-node based SSA-graph have been generated (right). The ϕ-nodes will there be the merging point for different definitions of values for a given variable.

2.2 χ-terms

A χ-function is a representation of how different control-flow options affect the value of a variable. For example, we can write down the value of variable b in block 7 in Figure 1 using χ-functions as $b = \chi^7(3, 4)$. Interpretation: variable b has the value 3 if

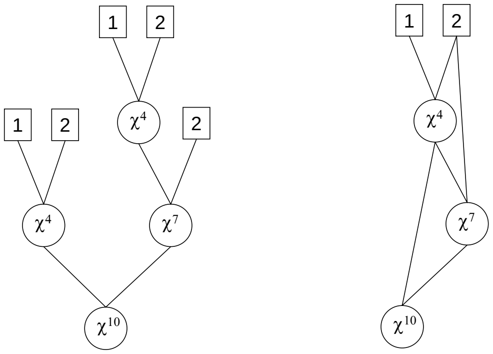


Fig. 2. Tree view of $\chi^{10}(\chi^4(1,2), \chi^7(\chi^4(1,2),2))$ and its graph representation.

it was reached from the first predecessor to block 7 in the control-flow graph, and the value 4 if it was reached from the second predecessor block. That is, a value expressed using χ-functions (a so-called χ-term) does not only contain all possible values, it also contains which control-flow options that generated each of these values.

The construction of the χ-term values and the numbering of the χ-functions is a part of a context-sensitive analysis. Every ϕ-node in an SSA graph represents a join point for several possible definitions of a single variable, say x. When the analysis reaches a block b containing a ϕ-node for x it “asks” all the predecessor blocks to give their definition of x and constructs a new χ-term $\chi^b(x_1, \ldots, x_n)$ where x_i is the χ-term value for x given by the i:th predecessor. If the i:th predecessor block does not define x by itself, it “asks” its predecessor for the value. This process continues recursively until each predecessor has presented a χ-term value for x. The process will terminate if any use of a value also has a corresponding definition.

Iteration in the code will generate loops in the control-flow and this need to be handled in the χ-term. Loop handling will be elaborated in section 5.4.

In summary, a χ-term is a composition of χ-functions and analysis values $a, b, \ldots \in V$. Each program p has a (possibly infinite) set of χ-functions $\mathcal{X}(p)$ and each χ-function $\chi_j^b \in \mathcal{X}(p)$ is identified by a pair (b, j) where the *block number* b indicates in what basic block its generating ϕ-node is contained, and the *iteration index* j indicates on what analysis iteration over block b the χ-function was generated.

Two χ-terms $\chi_i^b(x_1, \ldots, x_n)$ and $\chi_j^b(y_1, \ldots, y_n)$ from the same block b have the same *switching behavior* if they have the same iteration index (i.e. $i = j$). That is, for any execution of the program it holds that

$$\text{branch } x_k \text{ is selected } \Leftrightarrow \text{ branch } y_k \text{ is selected}$$

Thus, the switching behavior of a χ-function is determined by a pair (b, i) where b is the block number and i is the iteration index. In what follows, we will often skip the iteration index to simplify the notations. In these cases we assume that all involved χ-function have the same iteration index.

```
public int m(int a, int b) {
    if ( ... ) {
      a = a+1;
      b = b-1;
    }
    else {
      a = a-1;
      b = b+1;
    }
    return a + b;
}
```

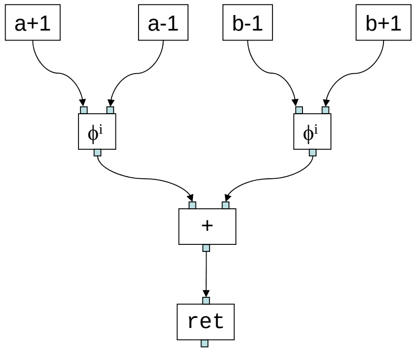


Fig. 3. An method with the corresponding SSA graph of the basic block containing the `return`-statement.

2.3 Tree and Graph Representation of χ-terms

Every χ- term can be naturally viewed as a tree. This is illustrated in Figure 2 (left) where we show the tree representation of the χ-term $\chi^{10}(\chi^4(1,2),\chi^7(\chi^4(1,2),2))$. Each edge represents a particular control flow option in this view and each path from the root node to a leaf value contains the sequence of control flow decisions required for that particular leaf value to come into play. The tree representation of a χ-term is easy to understand and important from theoretical point of view: many of the notations to be presented are easiest to understand in terms of operations on the tree representation.

To actually represent each χ-term as a tree is in practice much to costly. A more compact graph representation can easily be found by reusing identical subtrees, cf. Figure 2 (right), thus avoiding redundancies. This is the approach we recommend for any χ-term implementation and it is similar to how OBDDs are handled in [Bry92], and how classifier information is handled in [DLL14].

2.4 Operations on χ-terms and Shannon Expansions

The code fragment on the left-hand side of Figure 3 assigns different values to variables a and b in the two different branches of the `if`-statement and then returns the sum $a+b$. Using χ-terms, we can express the values of a and b as $\chi(a+1, a-1)$ and $\chi(b-1, b+1)$ respectively, and the sum $a+b$ as

$$+(\chi(a+1, a-1), \chi(b-1, b+1)).$$

Furthermore, we know that any execution takes either the first or the second branch of the `if`-statement (but we do not know which). This observation leads us to the following rewriting:

$$\begin{aligned} +(\chi(a+1,a-1),\chi(b-1,b+1)) &= \chi(\text{ first branch +, second branch +}) \\ &= \chi(+(a+1,b-1),+(a-1,b+1)) \end{aligned}$$

That is, we can make use of the fact that both χ-terms have the same switching behavior and apply the $+$ operator on each of the two branches separately before we merge the result. This rewriting can be seen as that we are pushing the $+$ operator one step closer to the leaf values.

Finally, we are now in a position where we can apply $+$ on a set of leaf values. In this case $+$ is well defined and we can fall back on ordinary integer arithmetics. This manipulation, where we also use the redundancy rule $\chi(t, t) = t$, can symbolically be written as:

$$\begin{aligned}\chi(+(a+1, b-1), +(a-1, b+1)) &= \chi((a+1)+(b-1), (a-1)+(b+1)) \\ &= \chi(a+b, a+b) = a+b\end{aligned}$$

The result indicates that no matter what branch of the `if`-statement we use, we will always get the result $a + b$. This simple example illustrates one of the strengths of using χ-terms, we can by using a few simple rewrite-rules make use of having stored flow-path information and "compute" more precise results than would have been possible in a context-insensitive approach.

That we can rewrite the addition of two χ-terms as a χ-term over the addition of a and b for each individual branch is in this case quite obvious. This rewrite rule for χ-term expressions is called a *Shannon expansion* [3]. It does not change the information represented by that term and leads therefore to an *equivalent* ($\equiv$) term. It may, however, change the size needed for representing a χ-term. For example,

$$\begin{aligned}t_a &= \chi^{10}(\chi^4(1,2), \chi^7(\chi^4(1,2), 2)) \\ &\equiv \chi^4(\chi^{10}(1, \chi^7(1,2)), \chi^{10}(2, \chi^7(2,2))) && \text{(expansion over } \chi^4\text{)} \\ &\equiv \chi^7(\chi^4(1,2), \chi^{10}(\chi^4(1,2), 2)) && \text{(expansion over } \chi^7\text{)}\end{aligned}$$

The Shannon expansion is also used to define the result of *applying* an operation to χ-terms. For instance, assume $t_a = \chi^{10}(\chi^4(1,2), \chi^7(\chi^4(1,2), 2))$ and $t_b = \chi^7(3,4)$, and assume further that $\texttt{apply}(+, i, j) = i + j$ for any two integers i and j. Then

$$\begin{aligned}\texttt{apply}(+, t_a, t_b) &= \texttt{apply}(+, \chi^{10}(\chi^4(1,2), \chi^7(\chi^4(1,2), 2)), \chi^7(3,4)) \\ &= \chi^{10}(\texttt{apply}(+, \chi^4(1,2), \chi^7(3,4)), \\ &\qquad \texttt{apply}(+, \chi^7(\chi^4(1,2), 2), \chi^7(3,4))) \\ &= \chi^{10}(\chi^7(\texttt{apply}(+, \chi^4(1,2), 3), \texttt{apply}(+, \chi^4(1,2), 4)), \\ &\qquad \chi^7(\texttt{apply}(+, \chi^4(1,2), 3), \texttt{apply}(+, 2, 4))) \\ &= \chi^{10}(\chi^7(\chi^4(\texttt{apply}(+, 1, 3), \texttt{apply}(+, 2, 3)), \\ &\qquad\quad \chi^4(\texttt{apply}(+, 1, 4), \texttt{apply}(+, 2, 4))), \\ &\qquad \chi^7(\chi^4(\texttt{apply}(+, 1, 3), \texttt{apply}(+, 2, 3)), 6)) \\ &= \chi^{10}(\chi^7(\chi^4(4,5), \chi^4(5,6)), \chi^7(\chi^4(4,5), 6))\end{aligned}$$

[3] The word "Shannon expansion" is taken from the OBDD literature [Bry92] where a similar procedure is used to rewrite boolean functions represented as OBDDs.

The first step above, expansion over χ^{10}, can be seen as if we push the operator + one step towards the leaf values by computing + for each one of the branches of χ^{10} individually, and then merge these values using χ^{10}. This process can be repeated until we reach the leaf values where the context-insensitive version of + can be applied.

This idea generalizes to any operator implementing a context-insensitive transfer function. For each context-insensitive operator $\tau : A \times B \times \ldots \times N \mapsto V$ their is a corresponding χ-induced operator $\widetilde{\tau} : X_A \times X_B \times \ldots \times X_N \mapsto X_V$ defined as $\widetilde{\tau}(t_a, \ldots, t_n) = \texttt{apply}(\tau, t_a, \ldots, t_n)$.

3 Structural Induction on χ-terms

Many basic properties such as commutativity and associativity of a context-insensitive operator τ are directly mapped to the χ-induced counterpart $\widetilde{\tau}$. In what follows, we will often want to proof statements like: Assume that statement $S(v_1, \ldots, v_n)$ is true for all abstract values $v_i \in V$. Then $\widetilde{S}(t_1, \ldots, t_n)$ (the χ-induced counterpart to S) is true for all χ-terms $t_i \in X_V$. A typical example of such a statement is:

Theorem 1. *Let $\tau : V \times V \mapsto V$ be a* commutative *operation and let $\widetilde{\tau} : X_V \times X_V \mapsto X_V$ be the corresponding χ-induced operator. It then holds that*

$$\widetilde{\tau}(t_a, t_b) = \widetilde{\tau}(t_b, t_a) \qquad \forall t_a, t_b \in X_V.$$

This type of statement can in many cases be proved by something called *structural induction* on χ-terms. This type of induction is similar in spirit to ordinary structural induction as presented in most text books (e.g. [AU95]). That is, we do an induction on the depth of χ-terms. The following is an outline for such induction proofs:

1. *The Base Case*: We show that $\widetilde{S}(v_a, \ldots, v_n)$ is true when $v_a, \ldots, v_n \in V \subseteq X_V$. This step can in most cases be done by applying the property $S(v_a, \ldots, v_n)$.
2. *The Inductive Hypothesis*: Let

   $$t_a = \chi^a(a_1, \ldots, a_p),\ \ldots, t_n = \chi^n(n_1,\ \ldots, n_q)$$

   and assume that

   $$\widetilde{S}(a_i, \ldots, n_j) \text{ is true } \forall (a_i, \ldots, n_j) \in \mathit{children}(t_a) \times \ldots \times \mathit{children}(t_n).$$

3. *The Inductive Step*: Prove that $\widetilde{S}(t_a, \ldots, t_n)$ is true using the assumptions in the inductive hypothesis and the definition of operations on χ-terms including Shannon expansion.

If we do so, then we can conclude that $\widetilde{S}(t_a, \ldots, t_n)$ is true for all χ-terms $t_a, \ldots, t_n \in X_V$. In order to exemplify this type of proof by structural induction we prove Theorem 1.

1. *The Base Case*: We are given that τ is commutative and we know that for every χ-induced operator it holds that

   $$\widetilde{\tau}(v_a, \ldots, v_n) = \tau(v_a, \ldots, v_n), \forall v_i \in V \subseteq X_V.$$

Thus, let $t_a = v_a, t_b = v_b$

$$\widetilde{\tau}(t_a, t_b) = \tau(v_a, v_b) = \tau(v_b, v_a) = \widetilde{\tau}(t_b, t_a), \qquad \forall v_a, v_b \in V \subseteq X_V.$$

This completes the base case.

2. *The Inductive Hypothesis*: Let $t_a = \chi^a(a_1, \ldots, a_p)$ and $t_b = \chi^b(b_1, \ldots, b_q)$ and assume that

$$\widetilde{\tau}(a_i, b_j) = \widetilde{\tau}(b_j, a_i) \qquad \forall (a_i, b_j) \in children(t_a) \times children(t_b).$$

3. *The Inductive Step*: Prove that $\widetilde{\tau}(t_a, t_b) = \widetilde{\tau}(t_b, t_a)$ is true using the assumptions in the inductive hypothesis. We start with the left-hand side:

$$\begin{aligned}
\widetilde{\tau}(t_a, t_b) &= \widetilde{\tau}(\chi^a(a_1, \ldots, a_p), \chi^b(b_1, \ldots, b_q)) \\
&= \chi^a(\widetilde{\tau}(a_1, \chi^b(b_1, \ldots, b_q)), \ldots, \widetilde{\tau}(a_p, \chi^b(b_1, \ldots, b_q)) \\
&= \chi^b(\chi^a(\widetilde{\tau}(a_1, b_1)), \widetilde{\tau}(a_2, b_1)), \ldots, \widetilde{\tau}(a_p, b_1)), \\
&\qquad\qquad \ldots, \\
&\qquad\quad \chi^a(\widetilde{\tau}(a_1, b_q)), \widetilde{\tau}(a_2, b_q)), \ldots, \widetilde{\tau}(a_p, b_q))) \\
&= \chi^b(\chi^a(\widetilde{\tau}(b_1, a_1)), \widetilde{\tau}(b_1, a_2)), \ldots, \widetilde{\tau}(b_1, a_p)), \\
&\qquad\qquad \ldots, \\
&\qquad\quad \chi^a(\widetilde{\tau}(b_q, a_1)), \widetilde{\tau}(b_q, a_2)), \ldots, \widetilde{\tau}(b_q, a_p)))
\end{aligned}$$

The first two rewritings are Shannon expansions in χ^a and χ^b, respectively. In the final rewriting we have used the inductive hypothesis $\widetilde{\tau}(a_i, b_j) = \widetilde{\tau}(b_j, a_i)$. Attacking the right-hand side with a similar approach (but no use of the inductive hypothesis) we find that:

$$\begin{aligned}
\widetilde{\tau}(t_b, t_a) &= \widetilde{\tau}(\chi^b(b_1, \ldots, b_q), \chi^a(a_1, \ldots, a_p)) \\
&= \chi^a(\widetilde{\tau}(\chi^b(b_1, \ldots, b_q), a_1), \ldots, \widetilde{\tau}(\chi^b(b_1, \ldots, b_q), a_p) \\
&= \chi^b(\chi^a(\widetilde{\tau}(b_1, a_1)), \widetilde{\tau}(b_1, a_2)), \ldots, \widetilde{\tau}(b_1, a_p)), \\
&\qquad\qquad \ldots, \\
&\qquad\quad \chi^a(\widetilde{\tau}(b_q, a_1)), \widetilde{\tau}(b_q, a_2)), \ldots, \widetilde{\tau}(b_q, a_p)))
\end{aligned}$$

Thus, $\widetilde{\tau}(t_a, t_b) = \widetilde{\tau}(t_b, t_a)$ and we have completed the proof of Theorem 1.

In what follows, we will not show any proofs that are straight forward applications of this type of structural induction since they pretty much look the same. For example, it is straight forward to show that properties like associativity, distributivity, and closure are conserved for any χ-induced operator $\widetilde{\tau} : X_V \times X_V \mapsto X_V$ by using a similar approach as in the previous proof.

4 The χ-term lattice

In Section 2, we learned that every context-insensitive operator τ has a χ-induced counterpart $\widetilde{\tau}$. This also holds for the abstract value lattice operators $\sqcap$ and $\sqcup$. The interesting point here is that these operators can be used to define a χ-induced χ-term lattice $\widetilde{\mathcal{L}}_V$ over the elements in X_V, the set of all χ-terms. This χ-induced lattice is important since it will be the value lattice in a χ-term based context-insensitive analysis.

Theorem 2. *For each lattice of abstract values* $\mathcal{L}_V = \{V, \sqcap, \sqcup, \top, \bot\}$ *there is a corresponding* χ*-induced lattice* $\widetilde{\mathcal{L}}_V = \{X_V, \widetilde{\sqcap}, \widetilde{\sqcup}, \top, \bot\}$ *where* $\widetilde{\sqcap}$ *and* $\widetilde{\sqcup}$ *are the* χ*-induced versions of* $\sqcap$ *and* $\sqcup$*.*

Showing that $\widetilde{\sqcap}$ and $\widetilde{\sqcup}$ have the properties *commutative*, *associative*, and *closure*, is a straight forward exercise in structural induction as presented in Section 3. The same holds for $t \,\widetilde{\sqcap}\, \bot = \bot$, $t \,\widetilde{\sqcup}\, \top = \top$ for all $t \in X_V$. We will not show it here.

Furthermore, we can use the χ-induced lattice operator $\widetilde{\sqcup}$ to define a partial ordering relation between χ-terms. The Connecting Theorem in [DP02] implies that

Theorem 3. *Let* $\widetilde{\mathcal{L}}_V = \{X_V, \widetilde{\sqcap}, \widetilde{\sqcup}, \top, \bot\}$ *be a* χ*-induced lattice for some abstract values* V *and let* $\widetilde{\sqsubseteq} : \widetilde{\mathcal{L}}_V \times \widetilde{\mathcal{L}}_V \mapsto \{\texttt{true}, \texttt{false}\}$ *be an operator defined as:*

$$t_1 \,\widetilde{\sqsubseteq}\, t_2 \iff t_1 \,\widetilde{\sqcup}\, t_2 = t_2, \qquad \forall t_1, t_2 \in X_V.$$

Then $\widetilde{\mathcal{P}}_V = \{\widetilde{\sqsubseteq}, X_V\}$ *is a (*χ*-induced) partial ordering over* X_V*.*

Due to the iteration indicis the χ-induced lattice $\widetilde{\mathcal{L}}_V$ has an infinite height. Thus implying that a data-flow analysis based on this lattice will not be guaranted to terminate. Further approximations are needed (cf. Section 5).

Theorem 4. *Let* $\tau : \mathcal{L}_A \mapsto \mathcal{L}_B$ *be a monotone function and let* $\widetilde{\tau} : \widetilde{\mathcal{L}}_A \mapsto \widetilde{\mathcal{L}}_B$ *be the corresponding* χ*-induced operator. It then holds that*

$$t_1 \,\widetilde{\sqsubseteq}\, t_2 \;\Rightarrow\; \widetilde{\tau}(t_1) \,\widetilde{\sqsubseteq}\, \widetilde{\tau}(t_2), \qquad \forall t_1, t_2 \in \widetilde{\mathcal{L}}_A$$

Once again, the proof of this theorem is a straight forward exercise in structural induction as presented in Section 3. The only tricky part is the final induction step where we must show that $t_a \,\widetilde{\sqsubseteq}\, t_b \;\Rightarrow\; \widetilde{\tau}(t_a) \,\widetilde{\sqsubseteq}\, \widetilde{\tau}(t_b)$ for two arbitrary χ-terms $t_a = \chi^a(a_1, \ldots, a_p)$, $t_b = \chi^b(b_1, \ldots, b_q)$ given the induction hypothesis

$$a_i \,\widetilde{\sqsubseteq}\, b_j \;\Leftrightarrow\; \widetilde{\tau}(a_i) \,\widetilde{\sqsubseteq}\, \widetilde{\tau}(b_j), \qquad \forall (a_i, b_j) \in children(t_a) \times children(t_b).$$

This can be done in two steps:

1. Show that

$$t_a \,\widetilde{\sqcup}\, t_b \;= t_b \;\Rightarrow\; a_i \,\widetilde{\sqcup}\, b_j \;= b_j \qquad \forall (a_i, b_j) \in children(t_a) \times children(t_b).$$

   which corresponds to $t_a \,\widetilde{\sqsubseteq}\, t_b \;\Rightarrow\; a_i \,\widetilde{\sqsubset}\, b_j, \forall (a_i, b_j)$. This can be done by comparing $t_a \,\widetilde{\sqcup}\, t_b$ with t_b after both has been Shannon expanded over both χ^a and χ^b.
2. Show that

$$\begin{aligned} \widetilde{\tau}(a_i) \,\widetilde{\sqcup}\, \widetilde{\tau}(b_j) \;=\; & \widetilde{\tau}(b_j), \forall (a_i, b_j) \in children(t_a) \times children(t_b) \\ & \Rightarrow \widetilde{\tau}(t_a) \,\widetilde{\sqcup}\, \widetilde{\tau}(t_b) = \; \widetilde{\tau}(t_b). \end{aligned}$$

   which corresponds to $\widetilde{\tau}(a_i) \,\widetilde{\sqsubseteq}\, \widetilde{\tau}(b_j), \forall (a_i, b_j) \Rightarrow \widetilde{\tau}(t_a) \,\widetilde{\sqsubseteq}\, \widetilde{\tau}(t_b)$. This can be done by a Shannon expansion of $\widetilde{\tau}(t_a) \,\widetilde{\sqcup}\, \widetilde{\tau}(t_b)$ over both χ^a and χ^b followed by repeated use of the identities $\widetilde{\tau}(a_i) \,\widetilde{\sqcup}\, \widetilde{\tau}(b_j) = \; \widetilde{\tau}(b_j)$.

These two steps, together with induction hypothesis, prove the inductive step.

5 χ-term Approximations

In this section, we present two different approximations to the fully context-sensitive approach outlined above. We refer to these two approximations as the *finite k-approximation* and the *loop approximation*. In the end of this section (Section 5.4), we show how these two approximations can be used to handle loops. These two approximations will in Section 5.5 be merged to something we refer to as a k-approximated analysis. This type of analysis is parametrized by a single precision parameter k. However, we start by introducing the concept of $\widetilde{\sqcup}$-approximations.

5.1 $\widetilde{\sqcup}$-Approximations

The aim of this section is to show that we always can replace any subterm $\chi_j^b(t_1, \ldots, t_n)$ in a χ-term t with $\widetilde{\sqcup}(t_1, \ldots, t_n)$. Hence, given that we interpret the χ-terms as values in a data-flow analysis, we still maintain a conservative approach. We refer to this type of χ-term manipulations as $\widetilde{\sqcup}$-approximations.

Theorem 5. *For any χ-term $\chi(t_1, \ldots, t_n) \in \widetilde{\mathcal{L}}_V$ it holds that*

$$\chi(t_1, \ldots, t_n) \mathrel{\widetilde{\sqsubseteq}} \widetilde{\sqcup}(t_1, \ldots, t_n)$$

Proof: Using Theorem 3 as a definition for $\widetilde{\sqsubseteq}$, it is sufficient to verify that

$$\widetilde{\sqcup}(\chi(t_1, \ldots, t_n),\ \widetilde{\sqcup}(t_1, \ldots, t_n)) = \widetilde{\sqcup}(t_1, \ldots, t_n).$$

This can be done in a few steps starting with a Shannon expansion over χ and ending by applying the redundancy rule

$$\begin{aligned}\widetilde{\sqcup}(\chi(t_1, \ldots, t_n),\ \widetilde{\sqcup}(t_1, \ldots, t_n)) &= \chi(\widetilde{\sqcup}(t_1,\ \widetilde{\sqcup}(t_1, \ldots, t_n)), \ldots, \\ &\qquad \widetilde{\sqcup}(t_n,\ \widetilde{\sqcup}(t_1, \ldots, t_n))) \\ &= \chi(\widetilde{\sqcup}(t_1, \ldots, t_n), \ldots, \widetilde{\sqcup}(t_1, \ldots, t_n)) \\ &= \widetilde{\sqcup}(t_1, \ldots, t_n)\end{aligned}$$

Theorem 6. *Let $t_a = \chi_j^b(a_1, \ldots, a_n)$ and $t_b = \chi_j^b(b_1, \ldots, b_n)$ be two χ-terms with the same switching behavior where $a_i \mathrel{\widetilde{\sqsubseteq}} b_i, \forall i \in [1, n]$. It then holds that $t_a \widetilde{\sqsubseteq} t_b$.*

Proof outline: Start with a Shannon expansion of $t_a \mathbin{\widetilde{\sqcup}} t_b$ over χ_j^b and use $a_i \mathbin{\widetilde{\sqcup}} b_i = b_i, \forall i \in [1, n]$ to verify that $t_a \mathbin{\widetilde{\sqcup}} t_b = t_b$.

Definition 1 **($\widetilde{\sqcup}$-approximation).** *Let $t \in X_V$ be a χ-term and let $a = \chi_j^b(a_1, \ldots, a_n)$ be a subterm of t. The $\widetilde{\sqcup}$-approximation of t with respect to a, denoted t_a^*, is a new term where a in t is replaced by $\widetilde{\sqcup}(a_1, \ldots, a_n)$.*

This definition is easy to understand using the tree representation of t. It simply means that we have replaced the subtree rooted by the node $\chi_j^b(a_1, \ldots, a_n)$ with $\widetilde{\sqcup}(a_1, \ldots, a_n)$.

Theorem 7. *Let $t \in X_V$ be a χ-term and let a be a subterm of t then $t \mathrel{\widetilde{\sqsubseteq}} t_a^*$.*

Proof outline: This follows directly from the Theorems 5 and 6. Theorem 5 says that the replaced subterm is less then (or equal to) the new subterm. Theorem 6 says that this property is propagated to the root.

Theorem 7 is important since it tells us how to make conservative approximations of χ-terms. That is, we can in any phase of the analysis replace a χ-term $\chi(t_1, \ldots, t_n)$ with $\widetilde{\sqcup}(t_1, \ldots, t_n)$ and still maintain a conservative approach.

5.2 The Finite k-Approximation

The construction of new χ-terms is a part of the context-sensitive analysis. When the analysis reaches a ϕ-node in block b for a variable x, it constructs a new χ-term by composing χ^b with all possible values for x. The newly constructed χ-term embodies all control-flow options that might influence the value of x at that point. The size of the χ-term representing x grows larger (without upper limit) as the analysis proceeds and more and more control-flow options influences the value of x. This represents a fully context-sensitive analysis where the effect of every control-flow option for every variable is kept at all times. In this section, we will present an approximation of the fully context-sensitive analysis where we only keep track of the last k control-flow options that might influence the value of a variable[4]. More “remote" control-flow options are merged using the ordinary context-insensitive merge operator $\sqcup$.

The finite k-approximation of χ-terms is easy to understand using the tree representation $G_t = \{N, E, r\}$. Whenever a new χ-term t is generated we replace all χ-terms $t_{sub} = \chi_i^b(t_1, \ldots, t_n)$ in $subterms(t)$ that has $depth(t_{sub}, t) \geq k$ with $\sqcup(t_1, \ldots, t_n)$. The process starts in the leafs and proceeds towards the root node. The result is a new χ-term $t^{(k)}$ with $depth(t^{(k)}) \leq k$. The process is outlined in Algorithm 1 where we define a recursive function $\texttt{kApprox}(k, t)$ that transforms an input χ-term t into a k-approximated χ-term $t^{(k)}$.

Algorithm 1 $\texttt{kApprox}(k, t = \chi_j^b(t_1, \ldots t_n)) \mapsto t^{(k)}$

```
if $k = 0$ then
    $t^{(k)} = \texttt{collapse}(t)$
else
    for all $t_i \in children(t)$ do
        $t_i^* = \texttt{kApprox}(k-1, t_i)$
    end for
    $t^{(k)} = \chi_j^b(t_1^*, \ldots, t_n^*)$
end if
return $t^{(k)}$
```

The post-order traversal in kApprox guarantees that our approximation starts from the leafs and proceeds towards the root. Parts of the tree that has a depth greater than

[4] Other approaches to limit the size of the χ-terms are possible. We could, for example, limit the tree *size* (i.e. number of nodes) rather than the *depth*.

$$\begin{aligned} a &= \chi^3(\chi^1(1,2), \chi^2(\chi^1(3,4),2)) \\ a^{(2)} &= \chi^3(\chi^1(1,2), \chi^2(\{3,4\},2)) \\ a^{(1)} &= \chi^3(\{1,2\}, \{2,3,4\}) \end{aligned}$$

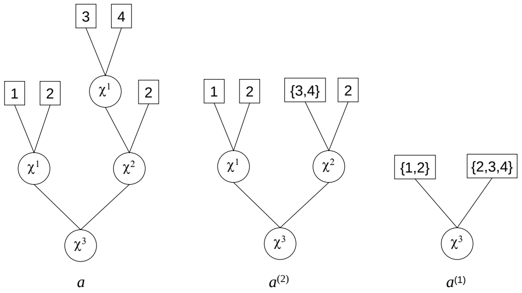


Fig. 4. Two different finite k approximations of the same χ-term a.

k is collapsed into leaf values by a process named `collapse` (see Algorithm 2). The process of merging leaf values proceeds until we have reached depth k of the input χ-term t. The result is a new χ-term $t^{(k)}$ that only embodies the last k control-flow options that might influence the value. Notice also that in the case $k = 0$ all context-sensitive information is lost and we have a context-insensitive analysis.

Algorithm 2 `collapse`$(t) \mapsto v$

```
if $t \in V$ then
    $v = t$
else
    let $v = \bot$
    for all $t_i \in children(t)$ do
        $v = v \sqcup \texttt{collapse}(t_i)$
    end for
end if
return $v$
```

Figure 4 shows the result of two different finite k approximations of the same χ-term a. On the left-hand side we have the result in print and on the right-hand side we have the same result depicted as an original tree and two trees where the depth have been reduced and the leaf values have been merged.

5.3 The Loop Approximation

According to Trapp at al. [THLL15] we know that the analysis of a loop will generate χ-terms like $x_n^b = \chi_n^b(\ldots\chi_{n-1}^b(\ldots)\ldots)$. That is, the newly created χ-term will have a subterm with the same block number and a lower iteration index. This pattern will probably occur over and over again since each loop iteration results in a new composition of χ^b with itself. This will result in χ-terms of infinite depth and a non-terminating analysis if no measure is taken to stop the iterations. In this section we will show one approximation that is a first step in that process. Informally, a χ-term $t = \chi_i^b(t_1, ..., t_n)$ is loop-approximated if every subterm of t that has the same block number as t is replaced by its $\widetilde{\sqcup}$- approximation.

Definition 2 (Loop-approximated χ-term). *A χ-term t is* loop approximated *if*

$$t_{sub} \in subterms(t) \ \wedge \ block(t_{sub}) = block(t) \ \Rightarrow t \rightarrow t^*_{t_{sub}}$$

where $t^*_{t_{sub}}$ is the $\widetilde{\sqcup}$-approximation of t with respect to t_{sub}. An analysis where every newly created χ-term is immediately loop approximated is said to be a loop approximated analysis.

This approximation is easy to understand as a tree manipulation. We make a post order traversal of the tree and replace each χ-term having the same block number as the root node with their context-insensitive approximation. This approach is outlined in Algorithm 3 where we define a recursive function `loopApprox`(t, b) that recursively visits all children before any merging takes place.

Algorithm 3 `loopApprox`$(t, b) \mapsto t^*$

```
for all $t_i \in children(t)$ do {Visit all children}
    $t_i^* =$ loopApprox$(t_i, b)$
end for
if $block(t) = b$ then
    let $t^* =$ apply$(\sqcup, t_1^*, \dots, t_n^*)$
    return $t^*$
else
    return $\chi^b(t_1^*, \dots, t_n^*)$
end if
```

The loop approximated analysis comes with a number of important observations:

1. Each χ-term will have a finite depth limited by the number of basic blocks that contain ϕ^b-nodes.
2. We can now drop the iteration index since only control-flow options from the last visit to any given block b will be recordered. Control-flow options from earlier visits have all been conservatively merged by $\widetilde{\sqcup}$- approximations. Thus, we will never have two χ-terms generated from the same block with different switching behavior.
3. If we drop the iteration index then the number of χ-functions at use will reduce to the number of basic blocks that contains a ϕ-node (a finite number).
4. A finite number of χ-functions and a finite depth of all χ-terms implies that we have a finite number of possible χ-terms. (Obvious if we think in terms of possible tree representations).

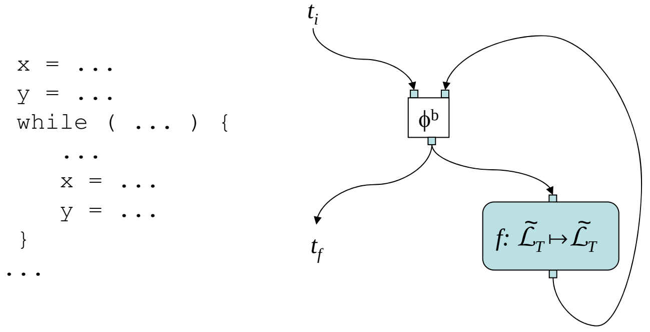
A consequence of the final observation is the following theorem:

Theorem 8. *The χ-induced lattice $\widetilde{\mathcal{L}}_V$ associated with a loop approximated analysis is a finite lattice with a finite height.*

One implication of this theorem is that every loop approximated analysis has a value lattice satisfying the ascending chain condition (see [DP02]) and that every analysis involving monotone transfer functions eventually will terminate (see [NNH99]). An example related to these properties is presented in the next section.

5.4 Analysis Loop Handling

The relation between the loop approximation and the loop handling is best illustrated with an example. In Figure 5, we show a hypothetical situation that illustrates a general case. On the left-hand side we have a piece of code that contains two variables x and y which values will be updated within the loop body. On the right-hand side we have the same situation depicted as an SSA graph. The two variable values entering the loop are represented as a tuple t_i and the updates within the loop body are represented by a mapping $f : \widetilde{\mathcal{L}}_T \mapsto \widetilde{\mathcal{L}}_T$. The loop approximated values generated by the ϕ-node in block b can then be written as



```
x = ...
y = ...
while ( ... ) {
   ...
   x = ...
   y = ...
}
...
```

t_i
ϕ^b
t_f
f: $\widetilde{\mathcal{L}}_T \mapsto \widetilde{\mathcal{L}}_T$

Fig. 5. A piece of source code and the corresponding graph. The mapping $f : \widetilde{\mathcal{L}}_T \mapsto \widetilde{\mathcal{L}}_T$ symbolizes the effect of the loop body on the variable tuple $[x, y]$.

$$\begin{aligned}
t_0 &= \chi^b(t_i, \bot) \\
t_1 &= \chi^b(t_i, f(t_i)) \\
t_2 &= \chi^b(t_i, \widetilde{\sqcup}(f(t_i), f^2(t_i))) \\
&\dots \\
t_n &= \chi^b(t_i, \widetilde{\sqcup}(f(t_i), f^2(t_i), \dots, f^n(t_i)))
\end{aligned}$$

where f^k is the composition of f with itself k times. Here t_i denotes the (loop approximated) value that t_f would get if we terminated the loop analysis after i iterations. We have derived these expressions by repeated use of $t_n = \chi^b(t_i, f(t_{n-1}))$ followed by

Shannon expansion of the “inner” χ^b, and a loop approximation. We have also assumed that $f(\bot) = \bot$. For example, t_1 was derived as

$$\begin{aligned} t_1 &= \chi^b(t_i, f(t_0)) = \chi^b(t_i, f(\chi^b(t_i, \bot))) \\ &= \chi^b(t_i, \chi^b(f(t_i), \bot)) \approx \chi^b(t_i, \widetilde{\sqcup}(f(t_i), \bot))) \\ &= \chi^b(t_i, f(t_i))) \end{aligned}$$

Notice also that we have dropped all iteration indices. This is possible in a loop approximated analysis where control-flow options from previous visits to a block b have all been merged by a $\widetilde{\sqcup}$-approximation.

This example shows that the effect of using a loop approximated analysis is that the analysis of loops (or any other cycle in the dependency graph) will generate a sequence of χ-terms all following a similar pattern:

$$t_n = \chi^b(t_i, T_n) \text{ where } T_n = \widetilde{\sqcup}(f(t_i), f^2(t_i), \ldots, f^n(t_i)).$$

The term T_n clearly represents a conservative approximation of the contribution from n loop iterations and $t_n = \chi^b_n(t_i, T_n)$ can be interpreted as: we go into the loop (T_n) or we do not (t_i).

Another consequence of using a loop approximated analysis is that

$$t_i, f(t_i), f^2(t_i), \ldots, f^n(t_i)$$

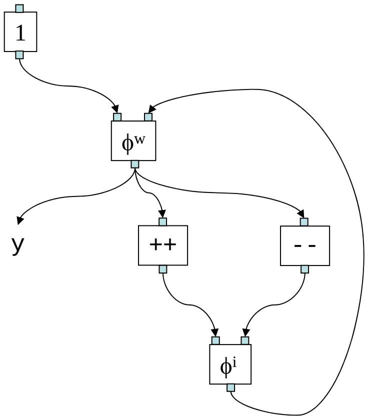
forms an ascending chain that will eventually get stabilized if f is a monotone function. Thus, after a finite number of loop iterations we will have $f^n(t_i) = f^{n-1}(t_i)$ and as result that $t_n = t_{n-1}$. This signals that the loop analysis can be terminated.

To gain a more concrete understanding of how the loop approximation can be used to terminate the analysis of a loop let us look into an intuitive example. If we have a `while`-loop that enclosed an `if`-statement that assigns a new value to a variable `x`. This situation is depicted in Figure 6 (SSA-graph left) where we also show the first three `x` values that might escape the loop (top right). The non-approximated values are given at the top of the figure and illustrates the problem of growth. That is, the set of control-flow options that might influence the value is growing larger and larger for each iteration. Furthermore, the values x^w_n and x^w_{n-1} returned from two consecutive iterations are not comparable (i.e. $x^w_n \not\mathrel{\widetilde{\sqsubseteq}} x^w_{n-1}$ and $x^w_{n-1} \not\mathrel{\widetilde{\sqsubseteq}} x^w_n$). This implies that we will never reach a stable situation where $x^w_n = x^w_{n-1}$ where we can terminate the loop analysis.

The situation is quite different in the loop approximated analysis of the loop where we after the second iteration get a result $x^w_2 = \chi^w(1, \top)$ that is not changed in the following iterations. (We have used a so-called “flat” lattice for integers where $n \sqcup \bot = n$ and $n \sqcup m = \top$ for any two lattice elements n and m, $n \neq m$. Moreover, we assume the following transfer functions for the $++(--)$ operations: $n++(--) = n+1(n-1)$ for integers n, $\top++(--) = \top$ and $\bot++(--) = \bot$. We have also removed redundant subterms.) The stable situation will get recognized by the analysis after the third iteration and the loop analysis will terminate.

5.5 The k-Approximated Analysis

In the previous section, we introduced two different approximations that make sense in almost any type of analysis. The loop approximation is necessary to guarantee analysis



No Approximations

$$\begin{aligned}
x_0^w &= \chi_0^w(1, \bot) \\
x_1^w &= \chi_1^w(1, \chi_0^i(\chi_0^w(2, \bot), \chi_0^w(0, \bot))) \\
x_2^w &= \chi_2^w(1, \chi_1^i(\chi_1^w(2, \chi_0^i(\chi_0^w(3, \bot), \chi_0^w(1, \bot))), \\
&\qquad \chi_1^w(0, \chi_0^i(\chi_0^w(1, \bot), \chi_0^w(-1, \bot)))))
\end{aligned}$$

Loop Approximated

$$\begin{aligned}
x_0^w &= \chi^w(1, \bot) \\
x_1^w &= \chi^w(1, \chi^i(2, 0)) \\
x_2^w &= \chi^w(1, \top) \\
x_3^w &= \chi^w(1, \top) \qquad \text{(Loop analysis terminated)}
\end{aligned}$$

Fig. 6. A loop approximated version of the given example . It illustrates how the loop approximation can be used to terminate the analysis of a loop.

termination and k in the finite k approximation is a precision parameter that can be seen as the size of "context memory" which decides how many previous control-flow options that each χ-term should try to remember.

In this section, we present notations and results that are valid for χ-terms, and analyses, that are both loop and finite k approximated. We will for simplicity refer to such χ-terms as *k-approximated* and an analysis that uses this approach will be called a *k-approximated analysis*.

In what follows, we will present results and notations related to a k-approximated analysis. This will be a rather brief presentation since many of the concepts has earlier on been introduced in a non-approximated version. However, this is the approach we intend to use in the rest of this section and the results presented here can be seen as the "final" results of this rather lengthy section.

The Normalized Set $X_V^{(k)}$ The set of all k-approximated χ-terms forms a subset of all χ-terms. We will here introduce a normalized form of this subset where we require *increasing* block numbers of the subterms along all leaf-to-root paths in a χ-term. (We assume that all leaf values $v \in V$ has been assigned the block number 0.) We refer to χ-terms having this specific structure as *normalized*. This approach of describing $X_V^{(k)}$ has the advantage that all χ-term values now has a unique χ-term (tree) representation. In what follows will take great care in maintaining this ordering.

Furthermore, we will assume that we have a *control-flow numbering* of the basic blocks in the flow-graph. That is, when numbering the basic blocks, we try to assign each basic block a higher number than their control-flow predecessors. More precisely, for any two blocks B^1 and B^2 we *try* to assign block numbers b^1 and b^2 such that:

$$B^2 \text{ always executed after } B^1 \Rightarrow b^2 > b^1.$$

We have here emphasized the word try since this type of block numbering is, although possible within a method, not possible for a whole program. However, it serves as a guide line for how to number the basic blocks in a program. The advantage of this approach is that subterms with a large depth can be considered as more "remote" than those closer to the root and that the finite k approximation can be defined more accurately.

ϕ-node Semantics A new χ^b-term is created whenever the analysis reaches a ϕ^b-node. The four steps involved in this process in case of k-approximated analysis are outlined in Algorithm 4. The first step is to remove every occurrence of χ^b in the input operands. This is done in `loopApprox` by replacing all subterms with block number b with their $\widetilde{\sqcup}$-approximation. Next, we construct a new χ-term that is now guaranteed to be loop approximated but neither normalized nor finite k approximated. The algorithm `normalize` takes care of the normalization (see Algorithm 5). It is a recursive process where we make repeated Shannon expansions over the χ-function with the highest block number. This process continues until we reach subterms having a block number that is less then b. Once `normalize` is applied we have a χ-term that is both loop approximated and normalized. We complete the ϕ-node handling by applying `kApprox` to make sure that the resulting χ-term has a maximum depth of k.

Algorithm 4 $\phi^{\mathtt{b}}_{\mathtt{op}}(t_1^{(k)}, \ldots, t_n^{(k)}) \mapsto t^{(k)}$

```
for all $t_i^{(k)} \in \{t_1^{(k)}, \ldots, t_n^{(k)}\}$ do
    $t_i = \mathtt{loopApprox}(t_i^{(k)}, b)$
end for
$t = \chi^b(t_1, \ldots, t_n)$
$t = \mathtt{normalize}(t)$
$t^{(k)} = \mathtt{kApprox}(t, k)$
```

Algorithm 5 $\mathtt{normalize}(t) \mapsto t^*$

```
$b = block(t)$
$max = maxBlock(children(t))$
if $max > b$ then
    for all $i \in [1, arity(\chi^{max})]$ do
        $t_i^* = \mathtt{normalize}(t|_{max:i})$
    end for
    $t^* = \chi^{max}(t_1^*, \ldots, t^*_{arity(\chi^{max})})$
    return $t^*$
else
    return $t$
end if
```

$\chi^{(k)}$-induced operators We have previously shown that for each context-insensitive operator $\tau : A \times B \times \ldots \times N \mapsto V$ their is a corresponding χ-induced operator $\widetilde{\tau} : X_A \times X_B \times \ldots \times X_N \mapsto X_V$. In the definition of $\widetilde{\tau}$, we used an algorithm `apply` that performed repeated Shannon expansions until it reaches the leaf values where the context-insensitive operator can be applied.

The definition of a $\chi^{(k)}$-induced operator can be seen as `apply` followed by a normalization procedure and a finite k-approximation. Using this approach, it is obvious that algebraic properties of a context-insensitive operator τ, like commutativity and associativity, are preserved for $\chi^{(k)}$-induced operators since we showed in Section 3 that they where preserved for any non-approximated χ-induced operator.

In Algorithm 6, we present a recursive algorithm `kPush` that performs all three activities (`apply`, normalization, and finite k cut-off) in a single traversal of the input operands.

Algorithm 6 `kPush`$(k, \tau, t_1, \ldots, t_n) \mapsto t^{(k)}$

```
$b = \max(block(t_1), \ldots, block(t_n))$
**if** $k = 0$ **then**
  **for all** $i \in [1, n]$ **do**
    $v_i = \texttt{collapse}(t_i)$
  **end for**
  $t^{(k)} = \tau(v_1, \ldots, v_n)$
**else if** $b = 0$ **then**
  $t^{(k)} = \tau(t_1, \ldots, t_n)$
**else**
  **for all** $i \in [1, arity(\chi^b)]$ **do**
    $c_i = \texttt{kPush}(k-1, \tau, t_1|_{b:i}, \ldots, t_n|_{b:i})$
  **end for**
  $t^{(k)} = \chi^b(c_1, \ldots, c_{arity(\chi^b)})$
**end if**
return $t^{(k)}$
```

The default handling in this algorithm is to push the operator τ towards the leaf values by making a Shannon expansion over the root χ-function in the operands that has the highest block number. This process guarantees that the result is normalized if all the input χ-terms are normalized.

The test $k = 0$ identifies the cut-off case where we have reached the maximum depth k of the resulting χ-term. In this case, we use `collapse` to make a conservative approximation of the remaining subtrees and apply the context-insensitive operator τ on the results. The case $b = 0$ identifies the case where all input operands are leaf values and the context-insensitive operator τ can be applied.

Finally, by using `kPush`, we can properly define the $\chi^{(k)}$-induced operators.

Definition 3. *For each context-insensitive operator $\tau : A \times B \times \ldots \times N \mapsto V$ their is a corresponding $\chi^{(k)}$-induced operator $\widetilde{\tau}^k : X_A^{(k)} \times X_B^{(k)} \times \ldots \times X_N^{(k)} \mapsto X_V^{(k)}$ defined* *as*

$$\widetilde{\tau}(t_a, \ldots, t_n) = \mathtt{kPush}(k, \tau, t_a, \ldots, t_n) \qquad \forall t_a, \ldots, t_n \in X_V^{(k)}.$$

We noted in Section 5.2 that Shannon expansions in combination with finite k-approximations are a bit problematic. The basic observation was that the interpretation of finite k approximations as an approach "where we only keep track of the last k control-flow options" was disturbed when Shannon expansions was used since they rewrite the structure of the χ-term. In this section, we have tried to minimize this problem by introducing a heuristic control-flow numbering of the basic blocks and introduced algorithms (`normalize` and `kPush`) that uses this block ordering to minimize the mixing of "remote" and "recent" control-flow options due to Shannon expansion.

The Lattice $\widetilde{\mathcal{L}}_V^{(k)}$ In Section 4, we introduced a χ-induced lattice $\widetilde{\mathcal{L}}_V$ that in general has an infinite height. In this section, we present the $\chi^{(k)}$-induced lattice $\widetilde{\mathcal{L}}_V^{(k)}$ which in contrast has a finite height. The finite height result follows from the fact that we always have a finite number of χ-terms in any loop approximated analysis. (see Section 5.3).

Theorem 9. *For each lattice of abstract values $\mathcal{L}_V = \{V, \sqcap, \sqcup, \top, \bot\}$ there is a corresponding $\chi^{(k)}$-induced lattice $\widetilde{\mathcal{L}}_V^{(k)} = \{X_V^{(k)}, \widetilde{\sqcap}^{(k)}, \widetilde{\sqcup}^{(k)}, \top, \bot\}$ where $\widetilde{\sqcap}^{(k)}$ and $\widetilde{\sqcup}^{(k)}$ are the $\chi^{(k)}$-induced versions of $\sqcap$ and $\sqcup$ defined in terms of the algorithm `kPush` as:*

$$\widetilde{\sqcup}^{(k)}(t_1, \ldots, t_n) = \mathtt{kPush}(k, \sqcup, t_1, \ldots, t_n)$$
$$\widetilde{\sqcap}^{(k)}(t_1, \ldots, t_n) = \mathtt{kPush}(k, \sqcap, t_1, \ldots, t_n).$$

That $\widetilde{\sqcap}^{(k)}$ and $\widetilde{\sqcup}^{(k)}$ are both commutative and associative follows from the preservation of algebraic identities previously discussed. The same holds for $t \mathbin{\widetilde{\sqcap}^{(k)}} \bot = \bot$, $t \mathbin{\widetilde{\sqcup}^{(k)}} \top = \top$ for all $t \in X_V^{(k)}$. Finally, closure follows from the design of `kPush` that guarantees to generate a new k-approximated χ-term.

We can use the $\chi^{(k)}$-induced lattice operators $\widetilde{\sqcap}^{(k)}$ and $\widetilde{\sqcup}^{(k)}$ to define a partial ordering relation between k-approximated χ-terms. The Connecting Theorem (see [DP02], page 39) implies that

Theorem 10. *Let $\widetilde{\mathcal{L}}_V^{(k)} = \{X_V^{(k)}, \widetilde{\sqcap}^{(k)}, \widetilde{\sqcup}^{(k)}, \top, \bot\}$ be a χ-induced lattice for some abstract values V and let $\widetilde{\sqsubseteq}^{(k)} : \widetilde{\mathcal{L}}_V^{(k)} \times \widetilde{\mathcal{L}}_V^{(k)} \mapsto \{\mathtt{true}, \mathtt{false}\}$ be an operator defined as:*

$$t_1 \mathbin{\widetilde{\sqsubseteq}^{(k)}} t_2 \iff t_1 \mathbin{\widetilde{\sqcup}^{(k)}} t_2 = t_2, \qquad \forall t_1, t_2 \in X_V^{(k)}$$

Then $\widetilde{\mathcal{P}}_V^{(k)} = \{\widetilde{\sqsubseteq}^{(k)}, X_V^{(k)}\}$ is a ($\chi^{(k)}$-induced) partial ordering over $X_V^{(k)}$.

To motivate the following result we can use the same line of arguments that we used when discussing the preservation of algebraic identities. That is, a $\chi^{(k)}$-induced operator $\widetilde{\tau}^{(k)}$ is just a finite k approximated χ-induced operator $\widetilde{\tau}$. From this it follows that

$$\widetilde{\tau}(t_1) \mathbin{\widetilde{\sqsubseteq}} \widetilde{\tau}(t_2) \Rightarrow \widetilde{\tau}^{(k)}(t_1) \mathbin{\widetilde{\sqsubseteq}^{(k)}} \widetilde{\tau}^{(k)}(t_2), \qquad \forall t_1, t_2 \in \widetilde{\mathcal{L}}_A^{(k)}$$

and consequently that

Theorem 11. *Let $\tau : \mathcal{L}_A \mapsto \mathcal{L}_B$ be a monotone function and let $\widetilde{\tau} : \widetilde{\mathcal{L}}_A^{(k)} \mapsto \widetilde{\mathcal{L}}_B^{(k)}$ be the corresponding $\chi^{(k)}$-induced operator. It then holds that*

$$t_1 \mathrel{\widetilde{\sqsubseteq}} t_2 \;\Rightarrow\; \widetilde{\tau}^{(k)}(t_1) \mathrel{\widetilde{\sqsubseteq}^{(k)}} \widetilde{\tau}^{(k)}(t_2), \qquad \forall t_1, t_2 \in \widetilde{\mathcal{L}}_A^{(k)}.$$

Thus, for any context-insensitive data-flow problem with value lattice $\mathcal{L}_V$ and transfer function $\tau^{(k)}$, we have $\chi^{(k)}$-induced counter parts $\widetilde{\mathcal{L}}_V$ and $\widetilde{\tau}^{(k)}$ that, since $\widetilde{\mathcal{L}}_V$ has a finite hight, is guaranteed to reach a fixed point.

Finally we know from the fundamental theory of data-flow frameworks [NNH99] that the time complexity for an analysis is proportional to the lattice height h^k. A rough estimate of h^k can be motivated as follows: let p be a program, let a be the maximum arity in any χ-function in $\mathcal{X}(p)$, and let h^v be the height of the context-insensitive value lattice $\mathcal{L}_V$. Furthermore, the tree representation of an arbitrary χ-term with depth k has about a^k leafs and the same number of subterms. Each leaf has a height of h^v. Thus, just by choosing different leaf values for this particular tree structure we can construct an ascending chain $x_1, \ldots, x_n$ that has length $O(a^k \cdot h^v)$. Furthermore, each element x_i in this chain can be further divided into an ascending subchain $x_{i1}, \ldots, x_{iN}$ by replacing each one of the a^k subterms by their $\widetilde{\sqcup}$-approximation. This gives this subchain a length of about a^k. Thus, a rough estimate of the maximum ascending chain length, and therefore the height h^k of the lattice $\widetilde{\mathcal{L}}_V^{(k)}$, is $O(a^k \cdot a^k \cdot h^v) = O(a^{2k} \cdot h^v)$.

6 Relation to Previous Work

As mentioned before, this paper is very much inspired by the ideas first presented by Martin Trapp in his dissertation [Tra99]. In that work, he presents an approach that is very similar to the k-approximated analysis. The major difference is that he presents his loop and finite k approximated approach as a *monolithic construct* without discussing the non-approximated case. To put it very short, he presents the set of normalized χ-terms $X_V^{(k)}$ together with rules for how to compute $\widetilde{\sqcup}$-approximations and $\chi^{(k)}$-induced operators $\widetilde{\tau}^{(k)}$. Furthermore, he states that $\chi(t_1, \ldots, t_n) \mathrel{\widetilde{\sqsubseteq}} \widetilde{\sqcup}(t_1, \ldots, t_n)$ and that we, in any phase of the analysis, can replace a χ-term $\chi(t_1, \ldots, t_n)$ with $\widetilde{\sqcup}(t_1, \ldots, t_n)$ and still maintain a conservative approach. The additional work that we have done is decribed in next section.

In [RKS99] and [KR00] Rütting et al. demonstrate an efficient and powerful approach by the usage of *value graphs*, which have initial similarities to our χ-terms representations. Both representations are based on a SSA representation of a program, and are using a graph representing the control flow in the program. The focus of their usage of *value graphs* is to find a solution for *Constant propagation* problem, while in our case we focus on value propagation for any data-flow analysis problem. Another difference is that we are using Shannon expansion to force our operators out to the leaves, where the operation can be evaluated. This is not the case in the *value graphs*, the operator nodes are scattered out in the *value graph*-representation, and therefore the evaluation of the result has another approach.

Lundberg and Löwe have in [LL13] been looking into the possibility of saving more information for doing a more precise points-to analysis. Their approach was to increase the call-depth ($k \geq 1$) for each new context. Their result of not getting any substantial precision improvement when the depth was increased ($k > 1$) implicates that increasing k in our *k-approximation* is not a sure way to get a significant precision improvement.

7 Summary and Future Work

Taken all together, our presentation of χ-terms is an attempt to verify many of the results that was presented, hinted, and implicitly assumed by Martin Trapp. It is also an attempt to verify (and understand) many of his “stated” results and definitions. In addition to this we have focused on the non-approximated χ-term expressions that we think is missing, and the reason for this is:

1. In order to properly motivate the introduction of the $\widetilde{\sqcup}$-approximation as a "conservative" approximation satisfying $t \mathrel{\widetilde{\sqsubseteq}} t_a^*$, for any $\widetilde{\sqcup}$-approximation t_a^* of a χ-term t, we need to introduce the non-approximated χ-term lattice $\widetilde{\mathcal{L}}_V$, and the corresponding partial ordering $\widetilde{\sqsubseteq}$. It is only in this context that we can verify that $t \mathrel{\widetilde{\sqsubseteq}} t_a^*$ (Theorem 7) and properly interpret a $\widetilde{\sqcup}$-approximation as "conservative".
2. We have been able to prove that many basic properties, such as commutativity and associativity, of a context-insensitive operator τ are directly mapped to the $\chi^{(k)}$-induced counterpart $\widetilde{\tau}^{(k)}$. We did this in two steps: i) We proved that it holds in the non-approximated case using structural induction, ii) We concluded that any identity that holds in the non-approximated case also must hold in the k-approximated case since the k-approximation is just a simple tree manipulation.
3. The two most important results are that any abstract value lattice $\mathcal{L}_V$ has a $\chi^{(k)}$-induced counterpart $\widetilde{\mathcal{L}}_V^{(k)}$, and that $\tau : \mathcal{L}_A \mapsto \mathcal{L}_B$ is monotone implies that $\widetilde{\tau}^{(k)} : \widetilde{\mathcal{L}}_A^{(k)} \mapsto \widetilde{\mathcal{L}}_B^{(k)}$ is monotone. These results make it possible to say that any context-insensitive data-flow framework has a $\chi^{(k)}$-induced context-sensitive counterpart.

The ideas from this paper will be used in future work to explore details about concrete context-sensitive dataflow problems such as constant folding and points-to analysis.

References

[AU95] A.V. Aho and J.D. Ullman. *Principles of Computer Science, (Second C edition)*. Computer Science Press, 1995.

[Bry92] R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[CFR$^+$91] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[DLL14] Antonina Danylenko, Jonas Lundberg, and Welf Löwe. Decisions: Algebra, implementation, and first experiments. *Journal of Universal Computer Science*, 20(9):1174–1231, September 2014.

[DP02] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order, Second Edition*. Cambridge University Press, 2002.

[KR00] Jens Knoop and Oliver Rüthing. Constant propagation on the value graph: Simple constants and beyond. In DavidA. Watt, editor, *Compiler Construction*, volume 1781 of *Lecture Notes in Computer Science*, pages 94–110. Springer Berlin Heidelberg, 2000.

[LL13] Jonas Lundberg and Welf Löwe. Points-to analysis: A fine-grained evaluation. *Journal of Universal Computer Science*, 18(20):2851 – 2878, 2013.

[NNH99] F. Nielsen, H.R. Nielsen, and C. Hankin. *Priciples of Program Analysis*. Springer, 1999.

[RKS99] Oliver Rüthing, Jens Knoop, and Bernhard Steffen. Detecting equalities of variables: Combining efficiency with precision. In Agostino Cortesi and Gilberto Filé, editors, *Static Analysis*, volume 1694 of *Lecture Notes in Computer Science*, pages 232–247. Springer Berlin Heidelberg, 1999.

[THLL15] Martin Trapp, Mathias Hedenborg, Jonas Lundberg, and Welf Löwe. Capturing and manipulating context-sensitive program information. *Software Engineering Workshops 2015*, 1337:154–163, 2015.

[Tra99] Martin Trapp. *Optimerung Objektorientierter Programme*. PhD thesis, Universität Karlsruhe, December 1999.

Approximating Context-Sensitive Program Information

Mathias Hedenborg[1], Jonas Lundberg[1], Welf Löwe[1], and Martin Trapp[2]

[1] Linnaeus University, Software Technology Group,
{Mathias.Hedenborg|Jonas.Lundberg|Welf.Lowe}@lnu.se
[2] Senacor Technologies AG, Martin.Trapp@senacor.com

Abstract. Static program analysis is in general more precise if it is sensitive to execution contexts (execution paths). In this paper we propose χ-terms as a mean to capture and manipulate context-sensitive program information in a data-flow analysis. We introduce *finite k-approximation* and *loop approximation* that limit the size of the context-sensitive information. These approximated χ-terms form a lattice with a finite depth, thus guaranteeing every data-flow analysis to reach a fixed point.

1 Introduction

Static program analysis is an important part of optimizing compilers and software engineering tools. These analyses predict properties of any execution of a given program, referred to as *program information*, by abstracting from its concrete execution semantics and its potential input values. Analyses can be context-sensitive or -insensitive, i.e., an analysis may or may not distinguish program information for different execution paths, i.e. for different *contexts*, e.g., the call contexts of a method. Context-sensitive analyses are, in general, more precise than their context-insensitive counterparts but also more expensive in terms of time and memory consumption.

In an iterative program, there are countably (infinitely) many contexts. Hence, merging the program information of some contexts is needed for the analysis to terminate. This, however, makes the analysis less context-sensitive, hence, less precise.

In Trapp et al. [THLL15], we focussed on capturing context-sensitive analysis information, i.e. contexts and program information for each program point, in a memory efficient way. In other words, we strived to delay merging the program information of different contexts for keeping analysis precision high. In the present paper, we discuss approximations that sacrifies precision for memory.

2 Background

This section introduces the notions and construction of χ-terms as components in static program analysis. Details are given in [THLL15].

2.1 SSA Representation

We assume the analysis to be based on a *Static Single Assignment* (SSA) graph representation [CFR$^+$91] of a program. Nodes in the SSA graph represent program points;

special ϕ-nodes represent merge points of the execution paths, i.e., contexts. Here we distinguish the program information of incoming paths by creating a χ-term connected to sub-terms, each representing the program information analyzed for the respective incoming execution path.

Figure 1 shows an example code with corresponding basic block and SSA-graph representations.

```
if (...)
  x = 1;
else
  x = 2
if (...){
  y = x;
  b = 3;
}
else {
  y = 2;
  b = 4;
}
if (...)
  a = x;
else
  a = y;
return a+b;
```

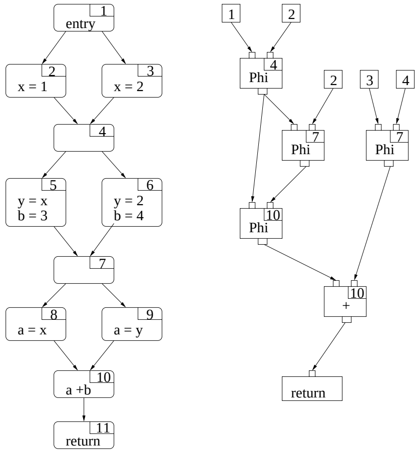


Fig. 1. A source code example with corresponding basic block and SSA graph structures.

In the figure the source code is transfered into numbered basic blocks (middle), and based on this a ϕ-node based SSA-graph have been generated (right). The ϕ-nodes will there be the merging point for different definitions of values for a given variable.

2.2 χ-terms

A χ-function is a representation of how different control-flow options affect the value of a variable. For example, we can write down the value of variable b in block 7 in Figure 1 using χ-functions as $b = \chi^7(3, 4)$. Interpretation: variable b has the value 3 if it was reached from the first predecessor to block 7 in the control-flow graph, and the value 4 if it was reached from the second predecessor block. That is, a value expressed using χ-functions (a so-called χ-term) does not only contain all possible values, it also contains which control-flow options that generated each of these values.

The construction of the χ-term values and the numbering of the χ-functions is a part of a context-sensitive analysis. Every ϕ-node in an SSA graph represents a join point for several possible definitions of a single variable, say x. When the analysis reaches a block b containing a ϕ-node for x it "asks" all the predecessor blocks to give their definition of x and constructs a new χ-term $\chi^b(x_1, \ldots, x_n)$ where x_i is the χ-term value for x given by the i:th predecessor. If the i:th predecessor block does not define x

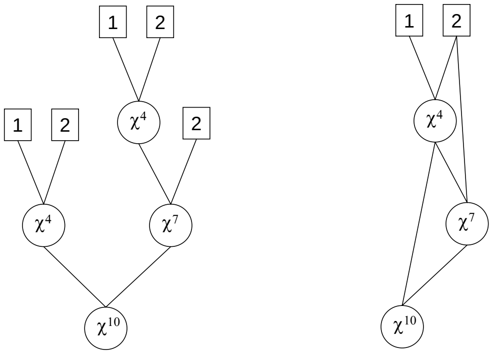


Fig. 2. Tree view of $\chi^{10}(\chi^4(1,2), \chi^7(\chi^4(1,2),2))$ and its graph representation.

by itself, it "asks" its predecessor for the value. This process continues recursively until each predecessor has presented a χ-term value for x. The process will terminate if any use of a value also has a corresponding definition.

In summary, a χ-term is a composition of χ-functions and analysis values $a, b, \ldots \in V$. Each program p has a (possibly infinite) set of χ-functions $\mathcal{X}(p)$ and each χ-function $\chi_j^b \in \mathcal{X}(p)$ is identified by a pair (b, j) where the *block number* b indicates in what basic block its generating ϕ-node is contained, and the *iteration index* j indicates on what analysis iteration over block b the χ-function was generated.

2.3 Tree and Graph Representation of χ-terms

Every χ-term can be naturally viewed as a tree. This is illustrated in Figure 2 (left) where we show the tree representation of the χ-term $\chi^{10}(\chi^4(1,2), \chi^7(\chi^4(1,2),2))$. Each edge represents a particular control flow option in this view and each path from the root node to a leaf value contains the sequence of control flow decisions required for that particular leaf value to come into play. A more compact graph representation (DAG) can easily be found by reusing identical subtrees, cf. Figure 2 (right), thus avoiding redundancies.

3 χ-term Approximations

In this section, we present two different approximations to the context-sensitive approach outlined above. We refer to these two approximations as the *finite k-approximation* and the *loop approximation*.

3.1 The Finite *k*-Approximation

The construction of new χ-terms is a part of the context-sensitive analysis. When the analysis reaches a ϕ-node in block b for a variable x, it constructs a new χ-term by composing χ^b with all possible values for x. The newly constructed χ-term embodies all control-flow options that might influence the value of x at that point. The size of the χ-term representing x grows larger (without upper limit) as the analysis proceeds and more and more control-flow options influences the value of x. The finite k-approximation of χ-terms can be seen as on operation on the tree representation $G_t = \{N, E, r\}$. Whenever a new χ-term t is generated we replace all χ-terms $t_{sub} = \chi_i^b(t_1, \ldots, t_n)$ in $subterms(t)$ that has $depth(t_{sub}, t) \geq k$ with $\sqcup(t_1, \ldots, t_n)$, where $\sqcup$ is the union operator on the realted value lattice. The use of k means that the last k analysis steps have had influences on the current value. The process starts in the leafs and proceeds towards the root node. The result is a new χ-term $t^{(k)}$ with $depth(t^{(k)}) \leq k$.

3.2 The Loop Approximation

According to Trapp at al. [THLL15] we know that the analysis of a loop will generate χ-terms like $x_n^b = \chi_n^b(\ldots \chi_{n-1}^b(\ldots) \ldots)$. That is, the newly created χ-term will have a subterm with the same block number and a lower iteration index. This pattern will probably occur over and over again since each loop iteration results in a new composition of χ^b with itself. This will result in χ-terms of infinite depth and a non-terminating analysis if no measure is taken to stop the iterations. Informally, a χ-term $t = \chi_i^b(t_1, ..., t_n)$ is loop-approximated if every subterm of t that has the same block number as t is replaced by its context-insensitive approximation.

4 Result

In the previous section, we introduced two different approximations that make sense in almost any type of analysis. The loop approximation is necessary to guarantee analysis termination and k in the finite k-approximation is a precision parameter that can be seen as the size of “context memory” which decides how many previous control-flow options that each χ-term should try to remember.

By using both these approaches in the analysis phase we can handle the need of extra information to meet the demands for context sensitivity and the precision in the result of the program analysis.

In the full paper we present: a) formal definitions of both k- and loop-approximations, b) efficient algorithms for both, c) proofs showing that approximated χ-terms forms a finite value lattice (depth k) guaranteeing each analysis to reach a fixed point.

References

[CFR+91] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[THLL15] Martin Trapp, Mathias Hedenborg, Jonas Lundberg, and Welf Löwe. Capturing and manipulating context-sensitive program information. *Software Engineering Workshops 2015*, 1337:154–163, 2015.

Objektorientierte Programmierung mit MOSTflexiPL

Christian Heinlein

Hochschule Aalen – Technik und Wirtschaft
christian.heinlein@hs-aalen.de

Abstract. MOSTflexiPL (Modular, Statically Typed, Flexibly Extensible Programming Language, vgl. http://flexipl.info) ist eine statisch typisierte Programmiersprache, deren Syntax vom Anwender nahezu beliebig erweitert und angepasst werden kann. Aufbauend auf einer kleinen Menge vordefinierter Grundoperatoren, können nach Belieben weitere Operatoren für unterschiedlichste Zwecke definiert werden. Da Operatoren beliebig viele Namen besitzen und auf beliebig viele Operanden angewandt werden können, decken sie neben den üblichen Präfix-, Infix- und Postfix-Operatoren auch Mixfix-Operatoren, Kontrollstrukturen, Typkonstruktoren und Deklarationsformen ab. Die Menge der vordefinierten Grundkonstrukte ist zwar klein, aber sehr ausdrucksstark. Um dies zu belegen, wird in diesem Beitrag gezeigt, wie man eine vollwertige objektorientierte Programmiersprache mit Vererbung, Untertyp-Polymorphie und dynamischem Binden in Form von MOSTflexiPL-Syntaxerweiterungen definieren kann. Tatsächlich geht das Spektrum der Möglichkeiten sogar weit über gängige Sprachen hinaus: Neben einfacher Vererbung und dem üblichen „single dispatch“, lassen sich auch mehrfache Vererbung in unterschiedlichen „Spielarten“ sowie „multiple dispatch“ und „predicate dispatch“ realisieren. Außerdem sind Typen ohne besondere Anstrengung „offen“, d. h. sie können problemlos nachträglich und modular um weitere Obertypen, Attribute und Operationen erweitert werden.

1 Einleitung

MOSTflexiPL (Modular, Statically Typed, Flexibly Extensible Programming Language) ist eine statisch typisierte Programmiersprache, die vom Anwender nahezu beliebig erweitert und angepasst werden kann [He12, He14]. Basierend auf einer kleinen Menge vordefinierter Grundoperationen (z. B. für Arithmetik, Logik und elementare Kontrollstrukturen), können in der Sprache selbst nach Belieben neue Operatoren, Operatorkombinationen, Kontrollstrukturen, Typkonstruktoren und Deklarationsformen definiert werden. Die Grundidee besteht darin, jedes dieser syntaktischen Konstrukte als *Operator* aufzufassen, der beliebig viele *Namen* (Operatorsymbole) und *Operanden* in beliebiger Reihenfolge besitzen kann. Beispielsweise besitzt der Additionsoperator `•+•` zwei Operanden (symbolisiert durch `•`) und einen Namen `+`, der bei einer Anwendung des Operators zwischen den Operanden (infix) steht, z. B. `2 + 3`. Der aus der Mathematik bekannte Betragsoperator `|•|` hingegen besitzt einen Operanden und zwei (zufällig gleiche) Namen, die um den Operanden herum (zirkumfix) stehen, z. B. `|-5|`. Eine Fallunterscheidung kann z. B. als Operator `if•then•else•`

`end` mit drei Operanden und vier Namen oder auch als Operator `•?•:•` mit drei Operanden und zwei Namen (Fragezeichen und Doppelpunkt) definiert werden. Der Typkonstruktor `•[•]` zur Definition von Arraytypen besteht aus zwei Namen (öffnende und schließende eckige Klammer) und zwei Operanden, von denen der erste ein Typ und der zweite eine ganze Zahl sein muss, z. B. `int [10]`. Aber auch Prozeduren und Funktionen lassen sich als Operatoren definieren, z. B. `max (•,•)` (mit klassischer imperativer Syntax, bei der die Klammern und das Komma einfach weitere Namen des Operators sind) oder `print•` (mit moderner funktionaler Syntax).

Tatsächlich erfolgt die Definition von Operatoren sehr ähnlich wie die Definition von Funktionen in anderen Sprachen. Und da mit jedem neuen Operator „nebenbei“ auch ein neues syntaktisches Konstrukt definiert wird, werden Syntaxerweiterungen auf die gleiche Art und Weise erstellt wie gewöhnliche Programme, d. h. es gibt hierfür keinen separaten Spezialmechanismus mit eigenen Ausdrucksmitteln und Regeln.

Da jedes syntaktische Konstrukt durch einen Operator repräsentiert wird, stellt jede Verwendung eines Konstrukts eine Operatoranwendung, d. h. einen *Ausdruck* dar. Hierbei werden auch Konstanten und Variablen sowie Literale wie z. B. `0` oder `"abc"` als nullstellige Operatoren aufgefasst, deren Anwendung einfach den jeweiligen Wert liefert. Beispielsweise ist der Ausdruck `if x >= 0 then x else -x end` eine Anwendung des Operators `if•then•else•end` auf die Operanden `x >= 0` sowie `x` und `-x`. Der Teilausdruck `x >= 0` ist wiederum eine Anwendung des Operators `•>=•` auf die Operanden `x` und `0`, bei denen es sich um Anwendungen der nullstelligen Operatoren `x` (eine Konstante oder Variable) und `0` (ein Literal) handelt. Ebenso ist `-x` eine Anwendung des Operators `-•` auf den Operanden `x`.

Die Anwendungsmöglichkeiten von MOSTflexiPL sind vielfältig. In erster Linie ist es als Allzwecksprache (general purpose language) gedacht, mit der man je nach persönlicher Präferenz sowohl imperativ als auch funktional programmieren kann. Der entscheidende Unterschied und Vorteil gegenüber anderen Sprachen besteht darin, dass man bei Bedarf jederzeit syntaktische Erweiterungen vornehmen kann, um bestimmte Dinge einfacher, kürzer, „natürlicher“ oder verständlicher ausdrücken zu können. Derartige Erweiterungen können entweder ad hoc für ein einzelnes Programm definiert werden oder aber in wiederverwendbaren Operatorbibliotheken zusammengefasst werden, die auch anderen Benutzern zur Verfügung gestellt werden können. Da sich Spracherweiterungen sehr leicht definieren und auch wieder ändern lassen, kann MOSTflexiPL auch als Experimentierplattform für neue Sprachkonstrukte verwendet werden, beispielsweise für objektorientierte Programmierung wie in diesem Beitrag. Weil sich die Syntax der Sprache nicht nur erweitern, sondern auch beliebig verändern und einschränken lässt, kann MOSTflexiPL schließlich auch als Hilfsmittel zur Definition und Implementierung anwendungsspezifischer Sprachen (DSLs) verwendet werden.

Die vordefinierten Grundkonstrukte und -konzepte der Sprache (die zum Teil aus dem Vorgängerprojekt APPLEs [He07] stammen), insbesondere auch das vorhandene Typsystem mit beschränkter parametrischer Polymorphie sowie benutzerdefinierbaren impliziten Typumwandlungen, sind so allgemein und ausdrucksstark, dass nicht nur „syntaktischer Zucker“, sondern auch weitreichende „paradigmatische“ Erweiterungen definiert werden können, was im folgenden demonstriert werden soll. Die Tatsache, dass sich Benutzer der Sprache bei Bedarf nahezu alles Gewünschte selbst definieren (oder aus Bibliotheken importieren) können, ist auch der Grund, warum bestimmte, aus anderen Sprachen gewohnte „Bequemlichkeiten“ a priori nicht vorhanden sind.

Im nachfolgenden Abschnitt 2 werden einige typische Beispiele für Syntaxerweiterungen durch neue Operatoren vorgestellt und damit nebenbei wichtige Grundkonstrukte von MOSTflexiPL vorgestellt und erläutert. Abschnitt 3 zeigt dann exemplarisch einige wichtige Syntaxerweiterungen für objektorientierte Programmierung. Abschnitt 4 enthält eine kurze Diskussion der entwickelten Sprachkonstrukte, während Abschnitt 5 mit Zusammenfassung und Ausblick schließt. Weiterführende Informationen zu MOSTflexiPL sowie viele weitere Beispiele findet man auf http://flexipl.info.

2 Beispiele für Syntaxerweiterungen durch Operatoren

2.1 Einfache Operatordeklarationen

Die folgenden Zeilen zeigen eine einfache Operatordeklaration in MOSTflexiPL:

```
["n" : int]               Parameterliste
n "²" : int               Signatur und Resultattyp
{ n * n }                 Implementierung
```

Sie besteht aus einer *Parameterliste* in eckigen Klammern, einer *Signatur* vor dem Doppelpunkt, einem *Resultattyp* danach sowie einer *Implementierung* in geschweiften Klammern. Die Signatur besteht ihrerseits aus Parametern und Zeichenketten in Anführungszeichen und definiert die *Syntax* des Operators, d. h. die syntaktische Form seiner Anwendungen: Jeder Parameter ist ein Platzhalter für einen Operanden, d. h. für einen Teilausdruck mit entsprechendem Typ, während eine Folge beliebiger Zeichen in Anführungszeichen einen Namen des Operators darstellt, der bei Anwendungen des Operators genau so (allerdings ohne die Anführungszeichen) hingeschrieben werden muss. Demnach ist z. B. `(2+3)`^{2} eine korrekte Anwendung des gerade definierten Operators, weil `(2+3)` ein Teilausdruck mit Typ `int` und ^{2} der Name des Operators ist.

Die Parameterliste besteht aus *Parameterdeklarationen* (ggf. durch Strichpunkte getrennt), bei denen es sich ebenfalls um einfache Operatordeklarationen handelt, die (bei den hier betrachteten einfachen Beispielen) lediglich aus einer Signatur und einem Resultattyp bestehen. Daher ist jedes Auftreten des Parameters `n` in Wirklichkeit eine Anwendung des nullstelligen Operators mit Signatur `"n"` und Resultattyp `int`.

Die Implementierung schließlich ist ein beliebiger Ausdruck, dessen Typ mit dem Resultattyp übereinstimmen muss und durch dessen Auswertung das Ergebnis einer Operatoranwendung entsteht. Beispielsweise entsteht der Wert des Ausdrucks `(2+3)`^{2}, indem zunächst der Parameter `n` mit dem Wert des zugehörigen Operanden `(2+3)` (also 5) initialisiert wird und anschließend die Implementierung `n * n` ausgewertet wird, die in diesem Fall den Wert 25 liefert.

Ein weiterer Operator `|•|`, der den absoluten Betrag seines Operanden als Ergebnis liefert, kann wie folgt definiert werden:

```
["x" : int]
"|" x "|" : int
{ if x >= 0 then x else -x end }
```

Hier besteht die Implementierung aus einer Anwendung des vordefinierten Verzweigungsoperators `if•then•else•end`, der, abhängig vom Wahrheitswert seines ersten Operanden, entweder den Wert seines zweiten oder den seines dritten Operanden als Ergebnis liefert. Da die Signatur aus einem senkrechten Strich in Anführungszeichen, dem `int`-Parameter `x` und einem weiteren senkrechten Strich in Anführungszeichen besteht, sind `|0|` und `|2-5|` exemplarische Anwendungen des Operators.

Um mehrere Ausdrücke nacheinander auszuwerten, kann man sie mit dem vordefinierten Operator `•;•` verknüpfen, bei dessen Anwendung – wie bei jeder Operatoranwendung – zunächst seine beiden Operanden (von links nach rechts) ausgewertet werden und der als Ergebnis einfach den Wert seines rechten Operanden liefert.

2.2 Konstanten, Typen und Variablen

Für einen beliebigen Typ `T` definiert eine Deklaration der Gestalt `"x" : T` eine *eindeutige Konstante* des Typs `T`, d. h. einen nullstelligen Operator `x`, der bei jeder Anwendung denselben eindeutigen Wert liefert und daher auch als *statischer Operator* bezeichnet wird. (Operatoren mit Implementierung, wie z. B. `"random" : int { ...... }`, die prinzipiell bei jeder Anwendung einen anderen Wert liefern können, werden zur Unterscheidung als *dynamische Operatoren* bezeichnet.)

Wenn man für `T` den vordefinierten Metatyp `type` verwendet, z. B. `"Person" : type`, erhält man einen neuen Typ `Person`, der verschieden von allen anderen Typen ist. (Dementsprechend sind vordefinierte Typen wie z. B. `int` auch nichts anderes als solche typwertigen Konstanten.) Anschließend kann man eindeutige Werte des Typs wie z. B. `"p" : Person` definieren, die vergleichbar mit Objekten in anderen Sprachen sind.

Wenn eine Deklaration eines statischen Operators Parameter besitzt, z. B. `["T" : type] "List" T : type`, so liefert der dadurch definierte Operator `List•` für jeden Wert seines Parameters `T` einen anderen eindeutigen Wert, sodass z. B. `List int` und `List Person` verschiedene Werte des Typs `type`, d. h. verschiedene Typen sind. Umgekehrt bezeichnet `List int` natürlich jedesmal den gleichen Typ.

Allgemein liefert ein statischer Operator bei Anwendung auf die gleichen Parameterwerte also immer das gleiche Ergebnis (was für einen dynamischen Operator mit Implementierung wiederum nicht garantiert werden kann) und bei Anwendung auf unterschiedliche Werte unterschiedliche Ergebnisse. Daraus folgt, dass zwei *statische Ausdrücke*, d. h. Ausdrücke, die nur statische Operatoren enthalten, den gleichen Wert liefern, wenn sie *strukturgleich* sind, d. h. wenn es sich um Anwendungen desselben (statischen) Operators auf paarweise strukturgleiche Operanden handelt.

Aufgrund dieser für den Compiler wichtigen Eigenschaft, dürfen statische Operatoren als *Typkonstruktoren* verwendet werden, während dynamische Operatoren in Typausdrücken verboten sind. Tatsächlich sind Typen in MOSTflexiPL einfach als statische Ausdrücke mit Typ type definiert.

Der vordefinierte Typkonstruktor •? liefert zu jedem Typ T den zugehörigen *Variablentyp* T?, dessen Werte jeweils eindeutige Variablen mit *Inhaltstyp* T sind. Beispielsweise deklariert "i" : int? eine Variable mit Inhaltstyp int, d. h. eine eindeutige Speicherzelle zur Speicherung von int-Werten, deren Inhalt durch eine Zuweisung wie z. B. i = i + 1 verändert werden kann.

Eine parametrisierte Variablendeklaration wie z. B.

```
["p" : Person]
p "." "name" : string?
```

liefert für jeden Wert des Parameters p eine andere eindeutige Variable p.name, d. h. sie ordnet jeder Person eine Variable mit Inhaltstyp string zu, in der der Name der Person gespeichert werden kann:

```
p.name = "Heinlein";
print p.name
```

Auf diese Weise lassen sich indirekt Datenstrukturen definieren, die bei Bedarf modular um neue „Attribute" erweitert werden können (sog. *offene Typen* [He07]).

Mit den folgenden Operatoren •->• und •.• wird die Definition und Verwendung solcher Attribute noch weiter erleichtert:

```
["X" : type; "Y" : type]
X "->" Y : type;

["X" : type; "Y" : type; "x" : X; "a" : X -> Y]
x "." a : Y?
```

Für zwei beliebige Typen X und Y liefert der statische Operator •->• jeweils einen eindeutigen Typ X -> Y, der zur Repräsentation von Attributen des Typs X mit Zieltyp Y verwendet werden kann, z. B.:

```
"Date" : type;
"day" : Date -> int;
"month" : Date -> int;
"year" : Date -> int;

"dob" : Person -> Date
```

Für ein Objekt x eines beliebigen Typs X und ein Attribut a eines zugehörigen Attributtyps X -> Y liefert der statische Operator •.• jeweils eine eindeutige Variable x.a mit Inhaltstyp Y, die zur Speicherung des entsprechenden Attributwerts verwendet werden kann, z. B.:

```
"d" : Date;
d.day = 8; d.month = 2; d.year = 1965;
p.dob = d
```

Variablen, denen noch kein Wert zugewiesen wurde, enthalten den Sonderwert nil, der die Abwesenheit eines echten Werts anzeigt. Dementsprechend liefern Zugriffe auf Attribute, denen noch kein Wert zugewiesen wurde, ebenfalls nil.

2.3 Implizite Typumwandlungen

Wenn ein Operator genau einen Operanden und keinen Namen besitzt, definiert er in natürlicher Weise eine implizite Typumwandlung vom Typ seines Operanden in seinen Resultattyp, z. B.:

```
["d" : Date]
d : string
{ d.day ++ "." ++ d.month ++ "." d.year }
```

Aufgrund seiner besonderen syntaktischen Struktur kann dieser Operator – ohne weitere Eingabesymbole zu verbrauchen – auf jeden Teilausdruck mit Typ `Date` angewandt werden und liefert als Resultat einen Wert des Typs `string` (konkret z. B. `"8.2.1965"`). Da der Compiler grundsätzlich jeden anwendbaren Operator „ausprobiert“ und Ausdrücke, die nicht typkorrekt sind, wieder „aussortiert“, wird er diesen Operator letztlich immer genau dann verwenden, wenn an einer bestimmten Stelle eine Umwandlung von `Date` nach `string` erforderlich ist, z. B.:

```
"s" : string?;
s = p.dob
```

2.4 Virtuelle Operatoren

Wenn man `var T` als Synonym für Variablentypen `T?` (vgl. Abschnitt 2.2) verwenden möchte, kann man versuchen, den Operator `var` • wie folgt zu definieren:

```
["T" : type]
"var" T : type
{ T? }
```

Da Typen vollwertige Werte sind, wird man zur Laufzeit tatsächlich feststellen, dass Vergleiche wie `var int == int?` als Resultat `true` liefern. Trotzdem ist diese Definition von `var` • relativ nutzlos, weil der Compiler dynamische Operatoren mit Implementierung in Typausdrücken nicht akzeptiert (vgl. Abschnitt 2.2) und eine Deklaration der Art `"i" : var int` daher fehlerhaft wäre.

Damit der Operator `var` • wirklich nützlich ist, muss er wie folgt als *virtueller Operator* definiert werden:

```
["T" : type]
"var" T = T?
```

Allgemein besitzt die Deklaration eines virtuellen Operators ebenfalls eine Parameterliste in eckigen Klammern sowie eine anschließende Signatur (im Beispiel `"var" T`). Anstelle von Resultattyp und Implementierung folgt dann jedoch eine sog. *Realisierung* nach einem Gleichheitszeichen (im Beispiel `T?`). Der Resultattyp des Operators ergibt sich implizit aus dem Typ der Realisierung (im Beispiel lautet er `type`).

Die Anwendung eines virtuellen Operators unterscheidet sich syntaktisch nicht von der Anwendung eines anderen Operators. Beispielsweise sind `var int` und `var List int` korrekte Anwendungen des Operators `var`•, während `var 2` fehlerhaft ist, weil der Operand `2` nicht den geforderten Typ `type` besitzt.

Die einzige Besonderheit besteht darin, dass eine Anwendung eines virtuellen Operators, nachdem sie erfolgreich auf Typkorrektheit überprüft wurde, vom Compiler sofort durch die Realisierung des Operators *ersetzt* wird, in der die Parameter des Operators wiederum durch die entsprechenden Operanden ersetzt werden. Demnach wird ein Ausdruck wie `var int` sofort durch die Realisierung `T?` ersetzt, in der der Parameter `T` wiederum durch den Operanden `int` ersetzt wird, d. h. der endgültige Ausdruck lautet `int?`.

Da diese Ersetzung bereits zur Übersetzungszeit stattfindet, wird eine Deklaration der Art `"i" : var int` jetzt vom Compiler akzeptiert, weil der Teilausdruck `var int` sofort durch `int?` ersetzt wird und die Deklaration daher vollkommen gleichbedeutend mit `"i" : int?` ist.

Da `var int` „in Wirklichkeit“ also `int?` bedeutet und nur „scheinbar“ etwas anderes darstellt, wird `var`• als „virtueller“ Operator und `T?` als seine „Realisierung“ bezeichnet.

2.5 Benutzerdefinierte Deklarationsoperatoren

Eine weitere Kategorie von Operatoren, die virtuell definiert werden müssen, damit sie den gewünschten Effekt haben, sind Deklarationsoperatoren, d. h. Operatoren, bei deren Anwendung andere Operatoren deklariert werden.

Wenn man beispielsweise Variablen (ähnlich wie in C, C++ und Java) in der Form `int "i"` anstelle von `"i" : int?` deklarieren möchte, kann man hierfür den folgenden virtuellen Operator • • verwenden:

```
["T" : type; "name" : string]
T name =
name : T?
```

Eine Anwendung wie z. B. `int "i"` (die nur aus Operanden besteht, weil der Operator keine Namen besitzt) wird wiederum durch die Realisierung des Operators, d. h. durch den Ausdruck `name : T?` ersetzt, in dem die Parameter `name` und `T` durch die Operanden `"i"` bzw. `int` ersetzt werden, sodass schließlich der Ausdruck `"i" : int?` entsteht.

3 Syntaxerweiterungen für objektorientierte Programmierung

Im folgenden werden exemplarisch einige Spracherweiterungen vorgestellt, die es erlauben, mit MOSTflexiPL objektorientiert zu programmieren. Aus Platzgründen müssen zwar zahlreiche Details weggelassen werden, aber anhand der gezeigten Beispiele kann man trotzdem einen guten Eindruck von den Möglichkeiten der Sprache gewinnen.

3.1 Hilfsoperatoren

Für zwei beliebige Typen X und Y stellt <X,Y> ein eindeutiges Attribut mit Typ X -> Y dar (vgl. §2.2), das verwendet werden kann, um für jedes Objekt des Typs X einen Verweis auf ein Objekt des Typs Y zu speichern:

```
["X" : type; "Y" : type]
"<" X "," Y ">" : X -> Y
```

Für Objekte x und y mit beliebigen Typen X bzw. Y stellt x <-> y unter Verwendung der Attribute <X,Y> und <Y,X> eine bidirektionale Verbindung zwischen diesen beiden Objekten her:

```
["X" : type; "Y" : type; "x" : X; "y" : Y]
x "<->" y : bool
{
    x.<X,Y> = y;
    y.<Y,X> = x;
    true
}
```

Der Resultattyp bool und der Resultatwert true haben keine weitere Bedeutung. Sie werden nur gebraucht, weil ein Operator immer einen Resultattyp besitzen und einen Resultatwert liefern muss.

3.2 Offene Typen

Mit den in §2.2 definierten Operatoren •->• und •.• können Datentypen sehr bequem – und nachträglich erweiterbar – definiert und verwendet werden, z. B.:

```
"Person" : type;
"name" : Person -> string;

"Date" : type;
"day" : Date -> int;
"month" : Date -> int;
"year" : Date -> int;

"dob" : Person -> Date;

"p" : Person;
p.name = "Heinlein";

"d" : Date;
d.day = 8; d.month = 2; d.year = 1965;
p.dob = d
```

3.3 Untertypen

Wenn in einer objektorientierten Sprache ein Typ (z. B. `Citizen`) als Untertyp eines anderen (z. B. `Person`) definiert wird, hat dies u. a. folgende Auswirkungen:

1. Ein Objekt des Untertyps kann implizit in den Obertyp umgewandelt werden (implizite Aufwärtsumwandlung).
2. Damit kann ein Objekt des Untertyps überall verwendet werden, wo ein Objekt des Obertyps erwartet wird (Ersetzbarkeit).
3. Insbesondere können alle Attribute des Obertyps auch für Objekte des Untertyps verwendet werden (Vererbung).
4. Für ein Objekt des Obertyps kann überprüft werden, ob es sich eigentlich um ein Objekt des Untertyps handelt (dynamischer Typtest). Wenn dies der Fall ist, kann das Objekt explizit in den Untertyp umgewandelt werden (explizite Abwärtsumwandlung).

Die erste dieser vier Eigenschaften, aus der die nächsten beiden automatisch folgen, kann in MOSTflexiPL wie folgt durch eine implizite Typumwandlung nachgebildet werden (vgl. §2.3):

```
"Citizen" : type;
"state" : Citizen -> string;
"idno" : Citizen -> string;

["c" : Citizen]
c : Person
{
    if c.<Citizen,Person> then
        c.<Citizen,Person>
    else
        "p" : Person;
        p <-> c;
        p
    end
}
```

Hier wird `Citizen` zunächst als normaler offener Typ mit Attributen `state` und `idno` definiert. Der anschließend definierte Operator ermöglicht eine implizite Aufwärtsumwandlung eines `Citizen`-Objekts `c` in ein „assoziiertes“ `Person`-Objekt `c.<Citizen,Person>`. Falls dieses noch nicht existiert, wird es als neues Objekt `p` erzeugt und mit dem Objekt `c` verbunden. Damit kann ein `Citizen`-Objekt z. B. wie folgt erstellt werden:

```
"c" : Citizen
c.name = "Heinlein";
c.state = "Germany"
```

Der folgende Operator realisiert die vierte Eigenschaft, indem er zu einem Person-Objekt p entweder das assoziierte Citizen-Objekt p.<Person,Citizen> liefert, sofern dieses existiert, oder den Sonderwert nil (vgl. §2.2):

```
["p" : Person]
p "?" "Citizen" : Citizen
{
    p.<Person,Citizen>
}
```

Damit kann dieser Operator sowohl für dynamische Typtests als auch für Abwärtsumwandlungen verwendet werden, z. B.:

```
if p?Citizen then
    print p?Citizen.state
end
```

Da die Definition einer Vererbungsbeziehung immer nach dem gleichen Schema erfolgt, ist es zweckmäßig, sie wiederum syntaktisch zu „verpacken“, z. B. mit Hilfe eines virtuellen Operators •=>•, dessen Definition hier aus Platzgründen weggelassen wird und der dann einfach wie folgt verwendet werden kann:

```
"Man" : type;
"bearded" : Man -> bool;
Man => Person
```

Da die impliziten Umwandlungen, die durch den Operator •=>• definiert werden, bei Bedarf auch transitiv angewandt werden, funktionieren mehrstufige Untertypbeziehungen ganz genauso.

3.4 Mehrfachvererbung

Durch mehrfache Verwendung des Operators •=>• kann ein Typ auch mehrere Obertypen besitzen, z. B.:

```
"User" : type;
"username" : User -> string;
"password" : User -> string;

"Employee" : type;
Employee => Person;
Employee => User
```

Damit kann ein Objekt des Typs Employee je nach Bedarf sowohl in Person als auch in User umgewandelt und entsprechend verwendet werden:

```
"e" : Employee;
e.name = "Heinlein";
e.username = "cheinl"
```

Da die Typen `Man` und `Citizen` beide Untertypen von `Person` sind und damit sowohl `Man`- als auch `Citizen`-Objekte jeweils ein assoziiertes `Person`-Objekt besitzen, entsteht im folgenden Beispiel durch Mehrfachvererbung die unerwünschte Situation, dass ein `MaleCitizen`-Objekt zwei verschiedene (indirekt) assoziierte `Person`-Objekte besitzt und deshalb die Umwandlung von `MaleCitizen` nach `Person` mehrdeutig ist (vgl. Abb. 1 links):

```
"MaleCitizen" : type;
MaleCitizen => Man;
MaleCitizen => Citizen
```

Abb. 1 rechts zeigt die eigentlich gewünschte Rautenstruktur, bei der es zu einem `MaleCitizen`-Objekt nur *ein* assoziiertes `Person`-Objekt gibt und die Umwandlung von `MaleCitizen` nach `Person` dementsprechend eindeutig ist. Um dies zu erreichen, sind mehrere Maßnahmen erforderlich:

- Die beiden indirekten Umwandlungen von `MaleCitizen` über `Man` bzw. `Citizen` nach `Person` müssen verboten werden, was sich mit Hilfe von *Ausschlussdeklarationen* realisieren lässt.[1]
- Stattdessen muss eine direkte Umwandlung von `MaleCitizen` nach `Person` definiert werden (gestrichelter Pfeil in der Abbildung).
- Bei der Erzeugung der assoziierten Objekte zu einem `MaleCitizen`-Objekt muss darauf geachtet werden, dass das `Man`- und das `Citizen`-Objekt auf dasselbe `Person`-Objekt verweisen.

Die entsprechenden Definitionen, die im Detail etwas „verzwickt“ sind, können wiederum hinter einem Operator mit „schöner“ Syntax versteckt werden, der dann z. B. wie folgt verwendet werden kann:

```
MaleCitizen => { Man | Citizen } => Person
```

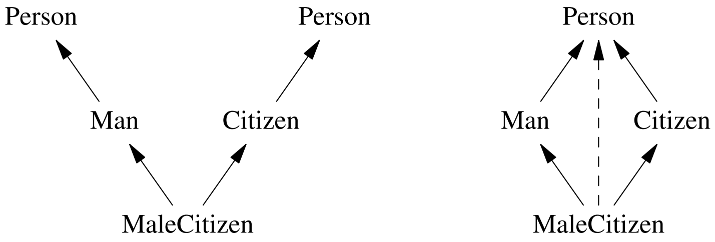


Abbildung 1: Mehrfachvererbung in V- bzw. Rautenform

[1] Ausschlussdeklarationen verbieten grundsätzlich bestimmte Verschachtelungen von Operatoranwendungen und werden primär zur Definition von Operatorvorrang eingesetzt. Um beispielsweise die bekannte Punkt-vor-Strich-Regel für arithmetische Operatoren zu implementieren, werden direkte Anwendungen der multiplikativen Operatoren `•*•` und `•/•` auf Anwendungen der additiven Operatoren `•+•` und `•-•` ausgeschlossen. Damit kann ein prinzipiell mehrdeutiger Ausdruck wie z. B. `a + b * c` nur noch als `a + (b * c)` interpretiert werden, weil die Interpretation als `(a + b) * c` ausgeschlossen ist.

Mit einer naheliegenden Verallgemeinerung dieses Operators lassen sich dann auch noch komplexere Vererbungsbeziehungen modellieren, z. B. Doppelstaatsbürger, die zwar zwei verschiedene `Citizen`-Teilobjekte, aber nur *ein* gemeinsames `Person`-Teilobjekt besitzen sollen (vgl. Abb. 2):

```
"Citizen1" : type;
"Citizen2" : type;
"DualCitizen" : type;

DualCitizen =>
    { Citizen1 => Citizen | Citizen2 => Citizen } => Person
```

Die Hilfstypen `Citizen1` und `Citizen2` werden gebraucht, um die beiden `Citizen`-Teilobjekte unterscheiden und ansprechen zu können, z. B.:

```
"dc" : DualCitizen;
"c1" : Citizen1 = dc; c1.state = "Germany";
"c2" : Citizen2 = dc; c2.state = "USA";
```

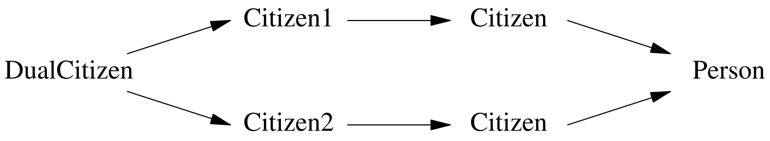


Abbildung 2: Doppelstaatsbürger

Durch mehrfache Anwendung dieser Operatoren lassen sich schließlich auch hochkomplexe Strukturen wie z. B. männliche oder weibliche Doppelstaatsbürger modellieren, die mehrere verschränkte Rauten enthalten.

3.5 Weiterführende Möglichkeiten

Ebenso wie man Attribute auch nachträglich zu offenen Typen hinzufügen kann, kann man mit den zuvor beschriebenen Operatoren auch Vererbungsbeziehungen nachträglich definieren – eine Möglichkeit, die objektorientierte Programmiersprachen normalerweise nicht bieten. Insbesondere ist es möglich, nachträglich Obertypen zu einem Typ hinzuzufügen.

Außerdem kann die Tatsache, dass ein Objekt eines Untertyps in Wirklichkeit aus mehreren miteinander verbundenen Teilobjekten besteht, ausgenutzt werden, um auf einfache Weise *dynamische Objektevolution* zu implementieren – eine Möglichkeit, die man in gängigen objektorientierten Programmiersprachen ebenfalls schmerzlich vermisst. Um beispielsweise nachträglich aus einer gewöhnlichen Person einen Mann oder einen Staatsbürger zu machen, genügt es, ein entsprechendes `Man`- oder `Citizen`-Objekt zu erzeugen und mit dem bereits vorhandenen `Person`-Objekt zu verbinden. Mit einer geeigneten syntaktischen Verpackung kann man dann als Anwender z. B. schreiben:

```
"p" : Person;
p.name = "Heinlein";

"c" : Citizen = p!Citizen;
c.state = "Germany"
```

3.6 Dynamisch gebundene Operationen

Dynamisch gebundene Operationen, die abhängig vom tatsächlichen Typ des Aufrufobjekts unterschiedliche Implementierungen ausführen, werden von Compilern üblicherweise durch „virtual function tables" implementiert. Wenn eine Programmiersprache Funktionszeiger o. ä. unterstützt, d. h. die Möglichkeit bietet, Funktionen o. ä. in Variablen zu speichern, können derartige Tabellen aber auch auf Anwendungsebene realisiert werden. Da Operatoren in MOSTflexiPL als Werte verwendet werden können, ist diese Möglichkeit gegeben, d. h. dynamisches Binden lässt sich prinzipiell auf Anwendungsebene implementieren.

Um nicht nur das gebräuchliche „single dispatch", sondern das wesentlich flexiblere „multiple dispatch" anbieten zu können, bei dem die dynamischen Typen *aller* Aufrufparameter bei der Auswahl der passenden Implementierung berücksichtigt werden können, sind komplexere Tabellenstrukturen erforderlich, die sich aber ebenfalls auf Anwendungsebene implementieren lassen.

Eine ansprechende syntaktische Verpackung könnte dann z. B. wie folgt aussehen:

```
"equal" ("p1" : Person; "p2" : Person) : bool
{
    p1.name == p2.name
};

"equal" ("c1" : Person?Citizen; "c2" : Person?Citizen) : bool
{
    c1.name == c2.name &
    c1.state == c2.state &
    c1.idno == c2.idno
}
```

Die erste Implementierung der Methode `equal` ist die allgemeinste, die für beliebige Personen `p1` und `p2` aufgerufen werden kann. Die zweite Implementierung stellt eine Spezialisierung dar, die nur ausgewählt wird, wenn beide Parameter den dynamischen Typ `Citizen` besitzen, d. h. `Person`-Objekte mit assoziierten `Citizen`-Objekten sind.

Wenn bei der Auswahl der passenden Methodenimplementierung nicht nur die Typen, sondern beliebige Eigenschaften der Aufrufparameter berücksichtigt werden können, spricht man von „predicate dispatching" [Er98]. Auch dieses Konzept lässt sich mit MOSTflexiPL prinzipiell umsetzen.

4 Diskussion

Das primäre Ziel dieses Beitrags ist es, die vielfältigen Möglichkeiten von MOSTflexiPL anhand einer praxisrelevanten „Aufgabenstellung“ – Unterstützung für objektorientierte Programmierung – zu demonstrieren. Da die konkrete „Lösung“ dieser Aufgabe, d. h. die exakte Syntax und Semantik der hierfür definierten Operatoren, für dieses Ziel sekundär ist, sollen weder die konkret gewählte Lösung noch mögliche Alternativen an dieser Stelle ausführlich diskutiert werden, sondern lediglich einige wesentliche Unterschiede zu „gängigen“ Ansätzen aufgezählt werden. Eine ausführlichere Diskussion findet sich in [He07], wo viele der hier vorgestellten Ideen bereits im Zusammenhang mit der Programmiersprache C+++ vorgestellt wurden.

- Auf das Konzept einer Klasse als feste Zusammenfassung einer Datenstruktur und der zugehörigen Operationen, wurde bewusst verzichtet.
- Stattdessen können Datentypen in beliebiger Reihenfolge definiert, mit Attributen versehen und in Vererbungsbeziehungen zueinander gesetzt werden. Damit sind nachträgliche Erweiterungen und Anpassungen viel leichter möglich als mit „starren“ Klassen.
- (Multi-)Methoden werden prinzipiell unabhängig von Klassen bzw. Typen definiert (ähnlich wie „generic functions“ in CLOS) und können daher ebenfalls problemlos nachträglich hinzugefügt werden. Damit existiert das für normale objektorientierte Sprachen schwerwiegende „expression problem“ [To04] in dieser Art schlicht und einfach nicht.
- Mehrfachvererbung wird ohne Einschränkungen unterstützt, weil sie für viele Anwendungen nützlich ist und das Prinzip von Vererbung und Untertyp-Polymorphie konsequent fortsetzt.
- Das dadurch unvermeidliche „diamond inheritance problem“ wird einfach, elegant und umfassend gelöst. Im Gegensatz zu anderen Sprachen mit Mehrfachvererbung (namentlich CLOS, Eiffel und C++), lassen sich beliebig komplexe Vererbungsstrukturen (wie z. B. männliche und weibliche Doppelstaatsbürger) ohne große Mühe modellieren.
- Ganz „nebenbei“ wird mit dynamischer Objektevolution auch noch ein Konzept unterstützt, das in statisch typisierten Programmiersprachen üblicherweise komplett fehlt.

Aus Platzgründen wurde das Prinzip des „information hiding“ [Pa72] komplett ausgeklammert. Aber auch hierfür bietet MOSTflexiPL mit sogenannten Sichtbarkeitsdeklarationen [He12] adäquate Unterstützung.

5 Zusammenfassung und Ausblick

In diesem Beitrag wurde gezeigt, wie man in MOSTflexiPL Operatoren zur Unterstützung objektorientierter Programmierung definieren und verwenden kann. Neben der hier vorgestellten „Lösung“ dieser „Aufgabe“, sind natürlich auch vielfältige alternative Lösungsansätze denkbar.

Ein essentielles Kernkonzept der Sprache, das zur Lösung der Aufgabe eingesetzt wird, sind benutzerdefinierbare implizite Typumwandlungen. Dieses muss im Detail noch etwas weiterentwickelt und verbessert werden, um beispielsweise unerwartete und unerwünschte Mehrdeutigkeiten aufgrund impliziter Umwandlungen zu eliminieren.

Eine weitere große „Baustelle“ von MOSTflexiPL ist nach wie vor die Ausgabe sinnvoller und hilfreicher Fehlermeldungen zur Übersetzungszeit sowie die Fortsetzung des Übersetzungsvorgangs nach einem Fehler. Außerdem muss für einen produktiven Einsatz von MOSTflexiPL einerseits die Effizienz des Compilers noch deutlich verbessert werden und andererseits das im Namen der Sprache bereits verankerte, aber momentan noch nicht verfügbare Modulkonzept implementiert werden.

Literaturverzeichnis

[Er98] M. Ernst, C. Kaplan, C. Chambers: “Predicate Dispatching: A Unified Theory of Dispatch.” In: E. Jul (ed.): *ECOOP’98 – Object-Oriented Programming* (12th European Conference; Brussels, Belgium, July 1998; Proceedings). Lecture Notes in Computer Science 1445, Springer-Verlag, Berlin, 1998, 186–211.

[He07] C. Heinlein: “Open Types and Bidirectional Relationships as an Alternative to Classes and Inheritance.” *Journal of Object Technology* 6 (3) March/April 2007, 101–151, http://www.jot.fm/issues/issue_2007_03/article3.

[He12] C. Heinlein: “MOSTflexiPL – Modular, Statically Typed, Flexibly Extensible Programming Language.” In: J. Edwards (ed.): *Proc. ACM Int. Symp. on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2012)* (Tucson, AZ, October 2012), 159–178.

[He14] C. Heinlein: “Fortgeschrittene Syntaxerweiterungen durch virtuelle Operatoren in MOSTflexiPL.” In: K. Schmid et al. (ed.): *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2014* (Kiel, February 2014). CEUR Workshop Proceedings, 193–212, http://ceur-ws.org/Vol-1129/paper4A.pdf.

[Pa72] D. L. Parnas: “On the Criteria to Be Used in Decomposing Systems into Modules.” *Communications of the ACM* 15 (12) December 1972, 1053–1058.

[To04] M. Torgersen: “The Expression Problem Revisited. Four New Solutions Using Generics.” In: M. Odersky (ed.): *ECOOP 2004 – Object-Oriented Programming* (18th European Conference; Oslo, Norway, June 2004; Proceedings). Lecture Notes in Computer Science 3086, Springer-Verlag, Berlin, 2004, 123–143.

Statische Typableitung für die optional typisierte Sprache Dart

Thomas S. Heinze, Anders Møller und Fabio Strocco

Aarhus Universitet, Institut for Datalogi, Åbogade 34, DK-8200 Aarhus N, Denmark
[t.heinze,amoeller,fstrocco]@cs.au.dk

1 Einführung und Motivation

Statisch typisierte Sprachen wie Java erleichtern die Programmierung durch die Möglichkeit, Programmierfehler bereits zur Übersetzungszeit aufzufinden, zu beheben und für die eigentliche Programmausführung weitestgehend auszuschließen. Zudem unterstützen die im Programm enthaltenen Typinformationen die Umsetzung hilfreicher Programmierwerkzeuge, wie beispielsweise Autovervollständigung und automatische Programmrestrukturierung. Gleichzeitig lassen sich die Typinformationen auch für die Übersetzung selbst, insbesondere zur Laufzeitoptimierung ausnutzen. Diesen Vorteilen statisch typisierter Sprachen stehen jedoch Einschränkungen hinsichtlich der Flexibilität der Programmierung gegenüber. Im Gegensatz dazu bieten die häufig zur Webprogrammierung genutzten dynamisch typisierten Programmiersprachen, namentlich JavaScript, eine höhere Flexibilität. Verbunden mit dieser ist nun aber das mögliche Auftreten von Typverletzungen zur Programmlaufzeit, sowie eine nur eingeschränkte Unterstützung von Programmierwerkzeugen und Programmoptimierungen.

Mit der Zunahme der Komplexität von Webanwendungen ergibt sich für die Webprogrammierung mit dynamischen Sprachen der Wunsch, auf die Vorteile statischer Typen zurückzugreifen, ohne dabei jedoch auf die Flexibilität der dynamischen Sprachen zu verzichten. Als ein Ansatz zur Verbindung der Vorteile von statischer und dynamischer Typisierung wurde die *optionale Typisierung* vorgeschlagen [2]. Diese erlaubt dem Programmierer selbst zu entscheiden, welche Teile eines Programms mit Typinformationen versehen werden, also statisch typisiert sind, und für welche Programmteile die Typen erst zur Laufzeit bestimmt werden sollen. Eine Umsetzung für ein solch optionales Typsystem bietet die Programmiersprache *Dart* [3]. In einem Dart-Programm kann einer Variablen entweder ein statischer Typ annotiert werden, oder aber die Variable wird als dynamisch typisiert ausgezeichnet und damit auch als zuweisungskompatibel zu jeder anderen Variablen, unabhängig von deren Typ. Verbunden mit der Möglichkeit zur selektiven Typisierung ist somit auch eine Lockerung der Typsicherheit eines Programms, da sich für die dynamisch typisierten Programmteile entsprechende Garantien nicht ohne Weiteres angeben lassen. Im Unterschied zu dem vergleichbaren Ansatz der graduellen Typisierung werden ferner auch keinerlei Garantien (etwa das Blame-Theorem [6]) zur Typsicherheit für die statisch typisierten Programmteile allein gegeben.

```
class Pair<E> {
    E left, right;
    void apply(Function f) {
        f(right); f(left);
    }
}
...
Pair<String> p =
        new Pair<String>();
//      new Pair<dynamic>();
Pair<Object> a = p;
a.left = 1;
print(p.left.length);
```

a) Kovarianz generischer Typen

```
class Type {}
class Subtype extends Type {
    bool field;
}
...
Pair<Type> p = new Pair<Type>();
p.right = new Subtype();
p.right.field = true;
p.left = new Subtype();
p.left.field = false;
p.apply((var e) {
    print(e.field);
});
```

b) Inkonsistente Typannotation

Abb. 1. Zwei Beispiele zur Verwendung generischer Typen in Dart: Sowohl in a) als auch in b) wird von der bestehenden Typprüfung kein Fehler angezeigt.

Um dennoch bereits während der Übersetzungszeit Aussagen zur Typsicherheit eines Dart-Programms treffen zu können, schlagen wir die Anwendung einer statischen Programmanalyse zur Typinferenz vor. Mit Hilfe einer solchen Analyse können sichere Abschätzungen zu den möglichen Typen in den dynamisch typisierten Programmteilen abgeleitet werden, die anschließend die Grundlage zur Prüfung der Typsicherheit bilden. Die Spracheigenschaften von Dart stellen dabei jedoch verschiedene Herausforderungen an Entwurf und Konzeption der Analyse, insbesondere im Hinblick auf die notwendigerweise *sichere* Ableitung der Typen, auf die im Folgenden überblicksweise eingegangen werden soll.

2 Optionale Typisierung in Dart

Die objektorientierte Programmiersprache Dart [3] wurde als Alternative zu JavaScript eingeführt, mit dem Ziel die Programmierung von Webanwendungen durch den optionalen Einsatz statischer Typen zu unterstützen. Dart weist Ähnlichkeiten mit statisch typisierten Sprachen auf, so erinnern Syntax und Klassenkonzept an Java. Im Gegenzug erlaubt der auch als Vorgabe verwendete Typ `dynamic` eine dynamische Programmierung analog JavaScript.

Dart-Programme lassen sich auf einer virtuellen Maschine ausführen oder werden nach JavaScript übersetzt. In Übereinstimmung mit dem Konzept der optionalen Typisierung wird dabei zwischen zwei verschiedenen Ausführungsmodi unterschieden. Im ersten *Produktionsmodus (Production Mode)* erfolgt die Programmausführung vollständig dynamisch und somit unabhängig von den im Programm deklarierten Typen. Im zweiten *Entwicklungsmodus (Checked Mode)* erfolgen hingegen eine Reihe von Typprüfungen auf Grundlage der vorhandenen Typannotationen. Zwischen zwei typbezogenen Laufzeitfehlern kann unterschieden werden. Einerseits führt der Zugriff auf eine nicht vorhandene Methode beziehungsweise auf ein nicht vorhandenes Feld zu einem Fehler (*Message Not Understood*). Andererseits liegt eine Typverletzung (*Subtype Violation*) vor, falls beispielsweise einem statisch typisierten Feld ein Wert außerhalb von dessen Definitionsbereich zugewiesen wird. Können erstgenannte Fehler in beiden Ausführungsmodi auftreten, beruhen letztgenannte Fehler auf den annotierten statischen Typen und sind somit auf den Entwicklungsmodus begrenzt.

Das Typsystem von Dart ist dabei bewusst als nicht korrekt entworfen worden [3]. So können etwa generische Klassen kovariant sein (vergleiche Reihungstypen in Java). In Abbildung 1 ist auf der linken Seite ein Programm mit einer Zuweisung zwischen kovarianten generischen Typen (`Pair<Object> a = p`) dargestellt. Offenbar führt die Programmausführung zu einem Laufzeitfehler, der aber von der bestehenden statischen Typprüfung trotz vollständig deklarierter statischer Typen nicht identifiziert wird. Im Produktionsmodus kommt es zum Fehler beim Zugriff auf das Feld `length` in der letzten Anweisung (*Message Not Understood*), im Entwicklungsmodus wird als Ursache dafür zumindest eine Typverletzung in Zuweisung `a.left = 1` signalisiert. Wird das Programm wie im Kommentar angegeben modifiziert, indem der dynamische Typ als Typargument verwendet wird, ist selbst das nicht mehr möglich.

3 Statische Typableitung für Dart

Eine Analyse zur statischen Typableitung muss diese Eigenschaften der Sprache Dart berücksichtigen. Insbesondere stellt sich die grundlegende Frage nach dem Umgang mit Typannotationen. Werden diese ignoriert, ergibt sich eine Typableitung in Übereinstimmung mit der Ausführung eines Programms im Produktionsmodus. Voraussetzung dafür ist jedoch das Vorliegen des vollständigen Programms. Sonst kann beispielsweise für die Parameter einer öffentlichen Methode keine Aussage zu den möglichen Typen getroffen werden. Entsprechend schwer gestaltet sich in diesem Fall die Analyse von Bibliotheken oder Programmteilstücken (siehe auch [1,4]). Im anderen Fall führt die Berücksichtigung von Typannotationen zu einer modularen Analyse. Anhand der für die Programmschnittstelle deklarierten statischen Typen sind nun Aussagen zu den einlaufenden Typen möglich – unter Annahme der Ausführung im Entwicklungsmodus, um die Korrektheit der Typannotationen zu gewährleisten. Gleichzeitig ist auch eine Reduktion des Analyseaufwands in Abhängigkeit vom Vorhandensein statischer Typannotationen zu erwarten, da zur Abschätzung des Typs eines Elements nun nicht mehr aufwändig der Objektfluss nachvollzogen werden muss, sondern direkt auf den deklarierten Typ zurückgegriffen werden kann.

Die von uns vorgeschlagene statische Analyse zur Typinferenz für Dart unterstützt beide Ansätze und damit sowohl den Produktions- als auch den Entwicklungsmodus. Die Typableitung beruht auf dem Verfahren der Zeigeranalyse [5], nur dass anstatt einer Überabschätzung für die Ziele von Zeigern eine Überabschätzung für die möglichen Typen abgeleitet wird. Zu diesem Zweck werden den Ausdrücken und Elementen eines Dart-Programms Typvariablen zugeordnet, die auf Mengen von Typen verweisen (vergleiche Funktion $[\![\,]\!]$ in

$Type = (ConcreteType \cup DeclaredType \cup ExternType) \setminus \{dynamic\}$

$[\![]\!]\colon Expression \cup Element \to \mathcal{P}(Type)$

$type\colon Element \to DeclaredType \cup ExternType$

$$\begin{array}{rc}
\text{Zuweisung } \texttt{x=e}: & type(\texttt{x}) \neq dynamic \Rightarrow \{type(\texttt{x})\} \subseteq [\![\texttt{x}]\!] \\
 & type(\texttt{x}) = dynamic \Rightarrow [\![\texttt{e}]\!] \subseteq [\![\texttt{x}]\!] \\
 & t \in [\![\texttt{e}]\!] \wedge t \in ExternType \Rightarrow \{t\} \subseteq [\![\texttt{x}]\!] \\
\text{Instanz } \texttt{new T()}: & \{T\} \subseteq [\![\texttt{new T()}]\!] \\
\texttt{literal} \text{ vom Typ } T: & \{T\} \subseteq [\![\texttt{literal}]\!] \\
\text{Feldzugriff } \texttt{x.f}: & t \in [\![\texttt{x}]\!] \wedge t' \leq: t \wedge type(t'\texttt{.f}) = dynamic \Rightarrow [\![t'\texttt{.f}]\!] \subseteq [\![\texttt{x.f}]\!] \\
 & t \in [\![\texttt{x}]\!] \wedge t' \leq: t \wedge type(t'\texttt{.f}) \neq dynamic \Rightarrow \{type(t'\texttt{.f})\} \subseteq [\![\texttt{x.f}]\!] \\
 & t \in [\![\texttt{x}]\!] \wedge t' \leq: t \wedge s \in [\![t'\texttt{.f}]\!] \wedge s \in ExternType \Rightarrow \{s\} \subseteq [\![\texttt{x.f}]\!] \\
 & t \in [\![\texttt{x}]\!] \wedge t \in ExternType \Rightarrow \{Object^E\} \subseteq [\![\texttt{x.f}]\!] \\
\text{Feldzugriff } \texttt{x.f=e}: & t \in [\![\texttt{x}]\!] \wedge t' \leq: t \wedge type(t'\texttt{.f}) = dynamic \Rightarrow [\![\texttt{e}]\!] \subseteq [\![t'\texttt{.f}]\!] \\
 & t \in [\![\texttt{x}]\!] \wedge t' \leq: t \wedge s \in [\![\texttt{e}]\!] \wedge s \in ExternType \Rightarrow \{s\} \subseteq [\![t'\texttt{.f}]\!]
\end{array}$$

Abb. 2. Auswahl von (vereinfachten) Regeln zur statischen Typableitung

Abbildung 2). Weiterhin werden Regeln zwischen den Typvariablen in Form von Teilmengenbeziehungen definiert. Als Lösung des dadurch charakterisierten Regelsystems ergibt sich eine konservative Abschätzung zu den Typen im Programm. Wie in Abbildung 2 ersichtlich, wird dabei zwischen *konkreten Typen* $t \in ConcreteType$ und *Deklarationstypen* $t \in DeclaredType$ unterschieden, da letztere auch alle im analysierten Programm deklarierten Untertypen $t' <: t$ umfassen. Eine dritte Kategorie bilden die *externen Typen*, wobei ein externer Typ $t \in ExternType$ ebenfalls seine Untertypen $t' <: t$ umfasst, nur das im Gegensatz zu den Deklarationstypen auch unbekannte Untertypen dazu zählen. Auf diese Weise sollen die von außerhalb einlaufenden Typen repräsentiert sein. Dies ist insofern wichtig, als dass die Methoden und Felder eines Typs in den von ihm abgeleiteten Untertypen überschrieben werden können. Da sich für die überschriebenen Felder und Methoden der dynamische Typ deklarieren lässt, folgt, dass etwa für den Feldzugriff über einen externen und damit unbekannten Typ keine Aussagen mehr zum Feldtyp möglich sind. Dies wird in Abbildung 2 durch gesonderte Regeln für den Fluss externer Typen modelliert.

Zur Unterstützung beider Ausführungsmodi sind die Regeln zur Typableitung in Abbildung 2 durch die Funktion *type* parametrisiert. Diese Funktion bildet für ein Programmelement, in Abhängigkeit von dessen deklariertem Typ, auf einen Deklarationstyp oder einen externen Typ ab. Grundlegende Idee ist nun, im Fall des Produktionsmodus für jedes Programmelement e unabhängig von dessen tatsächlich deklarierten Typ $type(e) = dynamic$ zu setzen. Im Fall des Entwicklungsmodus wird hingegen $type(e)$ auf den tatsächlich deklarierten Typ, oder für ein öffentliches Element auf den diesem entsprechenden externen Typ gesetzt. Grundsätzlich sind weitere Parametrisierungen möglich, etwa eine in der Typannotationen nur für öffentliche Elemente berücksichtigt werden.

Die Regeln entsprechen für den Produktionsmodus den Erwartungen (es gilt überall $\mathit{type}(e) = \mathit{dynamic}$). Im Wesentlichen werden die für Instanziierungsausdrücke und Literale erzeugten Typen, analog einer Zeigeranalyse, entlang des Datenflusses propagiert. Angewendet auf das Programmbeispiel auf der linken Seite von Abbildung 1 ergibt sich unter anderem: $\llbracket\texttt{new Pair<String>()}\rrbracket \subseteq \llbracket p\rrbracket$, $\{\mathit{Pair}^C\} \subseteq \llbracket\texttt{new Pair<String>()}\rrbracket$, $t \in \llbracket\texttt{a}\rrbracket \Rightarrow \llbracket\texttt{1}\rrbracket \subseteq \llbracket t\texttt{.left}\rrbracket$, $\llbracket\texttt{p}\rrbracket \subseteq \llbracket\texttt{a}\rrbracket$, $\{\mathit{int}^C\} \subseteq \llbracket\texttt{1}\rrbracket$, $t \in \llbracket\texttt{p}\rrbracket \Rightarrow \llbracket t\texttt{.left}\rrbracket \subseteq \llbracket\texttt{p.left}\rrbracket$. Die Lösung $\llbracket\texttt{p.left}\rrbracket = \{\mathit{int}^C\}$ erlaubt der folgenden Typprüfung für `print(p.left.length)` einen Fehler zu identifizieren, da der konkrete Typ int^C kein Feld `length` definiert (zur besseren Unterscheidung der Typkategorien verwenden wir Hochstellungen ^{C},^{D},^{E}).

Interessanter ist die Betrachtung der Typableitung für den Entwicklungsmodus. Vereinfachend soll für das Beispiel aus Abbildung 1 im Folgenden angenommen werden, dass keine öffentlichen Elemente definiert, und damit keine externen Typen abzuleiten sind. Wird, wie oben bereits angesprochen, in einem ersten Ansatz für jedes definierte Element e die Funktion $\mathit{type}(e)$ auf den jeweiligen deklarierten Typ gesetzt, ergibt sich für das Beispielprogramm unter anderem: $\{\mathit{Pair}\texttt{<}\mathit{Object}^D\texttt{>}^D\} \subseteq \llbracket\texttt{a}\rrbracket$, $\{\mathit{Object}^D\} \subseteq \llbracket\texttt{a.left}\rrbracket$. Mit dieser Lösung kann der sich für das Beispiel im Entwicklungsmodus ergebende Fehler (*Subtype Violation*) jedoch nicht nachvollzogen werden. Grund hierfür ist in der Kovarianz generischer Typen zu suchen (siehe auch Abschnitt 2), die für die Typannotation `Pair<Object>` von `a` berücksichtigt werden muss. Gleiches gilt, falls das modifizierte Beispiel betrachtet wird, indem `dynamic` an Stelle von `String` als Typargument auftritt. Auch dann kann der Typannotation, in diesem Fall `Pair<String>` von `p`, nicht vertraut werden, wobei der Grund nun nicht in der Kovarianz generischer Typen sondern im dynamischen Typargument liegt.

Um sichere Ergebnisse auch unter Berücksichtigung der optionalen Typisierung und des inkorrekten Typsystems von Dart zu ermöglichen, erfolgt eine Verfeinerung mit Hilfe einer weiteren Parametrisierungsfunktion *sound*. Anstatt für ein Programmelement e mit deklariertem statischen Typ ($\mathit{type}(e) \neq \mathit{dynamic}$) einfach nur diesen Typ zu nutzen, werden für "unsichere" Typen zusätzlich auch die entlang des Datenfluss propagierten Typen berücksichtigt, analog dem Vorgehen für den dynamischen Typ. Als Bedingung für die entsprechenden Regeln ergibt sich somit $\mathit{type}(e) = \mathit{dynamic} \vee \neg \mathit{sound}(\mathit{type}(e))$. Für das Beispielprogramm ergibt sich dieses Mal: $\{\mathit{Pair}\texttt{<}\mathit{String}^D\texttt{>}^C\} \subseteq \llbracket\texttt{new Pair<String>()}\rrbracket$, $\llbracket\texttt{p}\rrbracket \subseteq \llbracket\texttt{a}\rrbracket$, $\llbracket\texttt{new Pair<String>()}\rrbracket \subseteq \llbracket\texttt{p}\rrbracket, \{\mathit{Pair}\texttt{<}\mathit{Object}^D\texttt{>}^D\} \subseteq \llbracket\texttt{a}\rrbracket, \{\mathit{String}^D\} \subseteq \llbracket\texttt{a.left}\rrbracket$ $\{\mathit{Object}^D\} \subseteq \llbracket\texttt{a.left}\rrbracket$, $\{\mathit{Pair}\texttt{<}\mathit{String}^D\texttt{>}^D\} \subseteq \llbracket\texttt{p}\rrbracket$. Auf Grundlage dieser Typabschätzung kann die Typverletzung in `a.left = 1` identifiziert werden und analog auch der sich für das modifizierte Beispiel ergebende Laufzeitfehler.

Denkbar wäre hier ebenfalls gewesen, auf die für den Produktionsmodus abgeleiteten Typen zurückzugreifen, da diese zum gleichen Ergebnis der Typprüfung geführt hätten. Jedoch ist das nicht immer der Fall. Auf der rechten Seite von Abbildung 1 ist ein weiteres Dart-Programm angegeben. Betrachtet werden soll darin der Feldzugriff `e.field`, dabei handelt es sich bei `e` um einen dynamisch typisierten Parameter einer anonymen Funktion, die der Methode `apply` übergeben wird. Für dieses Beispiel tritt sowohl im Entwicklungs- als auch im Produktionsmodus kein Laufzeitfehler auf, da der Parameter `e` jeweils auf eine

Instanz der Klasse `Subtype` verweist. Der betrachtete Feldzugriff `e.field` erfolgt somit auf definierten Feldern. Allerdings wirft die gewählte Typannotation im Beispiel Fragen auf. Zwar entspricht der konkrete Typ der Felder `left` und `right` dem Typ `Subtype`, als deklarierter Typ ergibt sich aber über das Typargument der Instanz `Pair<Type>` der Typ `Type`. Da für diesen das Feld `field` nicht definiert ist, liegt eine zumindest im statischen Sinn inkonsistente Typdeklaration vor, die lediglich durch Verwendung des dynamischen Typs für den Parameter `e` geheilt wird. Situationen wie diese stellen keine unmittelbaren Fehler dar, können aber auf Entwurfsschwächen in einem Programm hinweisen und sollten sich daher ebenfalls identifizieren lassen. Dies ist aber nur möglich, falls die deklarierten Typen in die Typableitung mit einbezogen werden.

4 Zusammenfassung

Der vorliegende Beitrag beschreibt überblicksweise eine Analyse zur statischen Typableitung für die Sprache Dart. Die Eigenschaften von Dart, insbesondere die Möglichkeit zur optionalen Typisierung und das bewusst inkorrekt definierte Typsystem müssen bei Entwurf und Konzeption berücksichtigt werden und führen für die beschriebene Analyse zu einem parametrisierten Entwurf. Dieser gestattet sowohl die sichere Typableitung für ein vollständig vorliegendes Dart-Programm unter Annahme der Programmausführung im Produktionsmodus, als auch eine modulare Typableitung bei Berücksichtigung der in einem Programm enthaltenen Typannotationen analog des Entwicklungsmodus.

Literatur

[1] Allen, Nicholas ; Krishnan, Padmanabhan ; Scholz, Bernhard: Combining Type-Analysis with Points-To Analysis for Analyzing Java Library Source-Code. In: *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, June 14, 2015, Portland, OR, USA*, ACM, 2015, S. 13–18

[2] Bracha, Gilad ; Griswold, David: Strongtalk: Typechecking Smalltalk in a Production Environment. In: *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Eighth Annual Conference, Washington, DC, USA, September 26 – October 1, 1993, Proceedings*, ACM, 1993, S. 215–230

[3] *Dart Programming Language Specification.* Ecma Intl., Standard ECMA-408, 2015

[4] Rastogi, Aseem ; Chaudhuri, Avik ; Hosmer, Basil: The Ins and Outs of Gradual Type Inference. In: *POPL'12, Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 25-27, 2012, Philadelphia, PA, USA*, ACM, 2012, S. 481–494

[5] Sridharan, Manu ; Chandra, Satish ; Dolby, Julian ; Fink, Stephen J. ; Yahav, Eran: Alias Analysis for Object-Oriented Programs. In: *Aliasing in Object-Oriented Programming.* Springer, 2013 (LNCS 7850), S. 196–232

[6] Wadler, Philip ; Findler, Robert B.: Well-Typed Programs Can't Be Blamed. In: *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009, Proceedings*, Springer, 2009 (LNCS 5502), S. 1–16

The Symbolic Execution Debugger:
a Productivity Tool for Java Based on Eclipse and KeY

Martin Hentschel, Richard Bubel, and Reiner Hähnle

TU Darmstadt, Dept. of Computer Science, Darmstadt, Germany
hentschel|bubel|haehnle@cs.tu-darmstadt.de

Abstract. We present the Symbolic Execution Debugger (SED), an extension of the Eclipse debug platform for interactive symbolic execution. Being based on symbolic execution, its functionality goes beyond that of traditional interactive debuggers. For instance, debugging can start directly at any method or statement and all program execution paths are explored simultaneously. To support program comprehension, execution paths as well as intermediate states are visualized.
Using KeY as underlying symbolic execution engine, SED supports sequential JAVA programs and the inspection of verification proofs.

Keywords: Symbolic Execution, Debugging, Program Execution Visualization

1 Introduction

This updated and extended version of [7] presents the Symbolic Execution Debugger[1] (SED), a language independent extension of the Eclipse debug platform for symbolic execution. Symbolic execution [3,4,10,11] is a program analysis technique based on the interpretation of a program with symbolic values. This makes it possible to explore *all* concrete execution paths (up to a finite depth). We describe an SED implementation that uses KeY [2] as the underlying symbolic execution engine, supporting sequential JAVA without floats, garbage collection and dynamic class loading. Our main contributions are the SED platform, interactive symbolic execution of JAVA and visualization of program behavior including unbounded loops and method calls.

The SED supports traditional debugger functionality like step-wise execution or breakpoints, and enhances it as follows: Debugging can begin at any method or any other statement in a program, no fixture is required. The initial state can be specified partially or not at all. During symbolic execution all feasible execution paths are discovered, thus it is not necessary to set up a concrete initial program state leading to an execution where a targeted bug occurs. At any time each intermediate state can be inspected using the SED. Intermediate states

[1] The website www.key-project.org/eclipse/SED contains an Installation and User Guide (including instructions on how to use API classes), as well as a screencast and theoretical foundations.

tend to be small and simple, because symbolic execution can be started close to the suspected location of a bug and the symbolic states contain only program variables accessed during execution. This makes it easy for the bug hunter to comprehend intermediate states and the actions performed on them to find the origin of a bug. Heisenbugs [5], a class of program errors that disappear while debugging, are avoided as the behavior of a program is correctly reflected in its symbolic execution. Besides debugging the SED platform allows to visualize and explore results of static analysis based on symbolic execution.

2 Symbolic Execution

Symbolic execution (SE) means to execute a program with symbolic values in lieu of concrete values. We explain SE and how it is used interactively in the SED by example: method `eq` shown in the listing in Fig. 1 compares the given `Number` instance with the current one.

For a Java method to be executed it must be called explicitly. For instance, the expression **`new`** `Number().eq(`**`new`** `Number());` invokes `eq` on a fresh instance with a different instance as argument. This results in a single execution path: first the guard in line 5 is evaluated to true, as fields of integer type are initialized with 0 by default. Finally, true is returned as result. To inspect another execution path the method has to be called in a different state.

Let us execute method `eq` symbolically, i.e., without a concrete argument, but a reference to a symbolic value n which can represent any object or **`null`**. In our SE tree notation we use different icons to underscore the semantics of nodes. As Fig. 1 shows, the root is a *Start Node* representing the initial state and the program fragment (any method or any block of statements) to execute. Here a call to `eq` is represented by its *Method Call* child node.

The **`if`**-guard, represented as a *Branch Statement* node, splits execution when the field `value` is accessed on the symbolic object n. Because nothing is known about n, it could be **`null`**. The *Branch Condition* children nodes show the condition under which each path is taken. On the left, where n is not **`null`**, the comparison in the **`if`**-guard splits execution again. If both values are the same, the **`return`** statement is executed, indicated by a *Statement* node. Now the symbolic path of the method is fully executed and returns true in the *Method Return* child node. This SE path ends in the *Termination* node. The branch where the values are different looks similar, but false is returned instead. In the rightmost branch the parameter n has the value **`null`** and SE ends with an uncaught `NullPointerException`, visualized as an *Exceptional Termination* node.[2]

In contrast to concrete execution, SE does not require fixture code and discovers all feasible execution paths (up to its execution depth). Each SE path through an SE tree may represent infinitely many concrete executions and is characterized by its path condition (the conjunction of all branch conditions on

[2] The instantiation of the thrown exception is not visualized since we do not include execution of Java API methods for simplicity.

```
public class Number {
   private int value;

   public boolean eq(Number n) {
      if (value == n.value) { return true; }
      else { return false; }
   }

   // ...
}
```

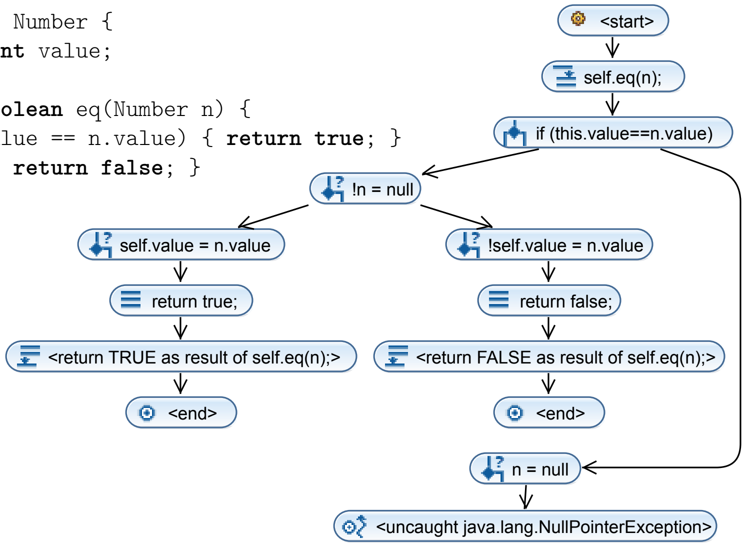


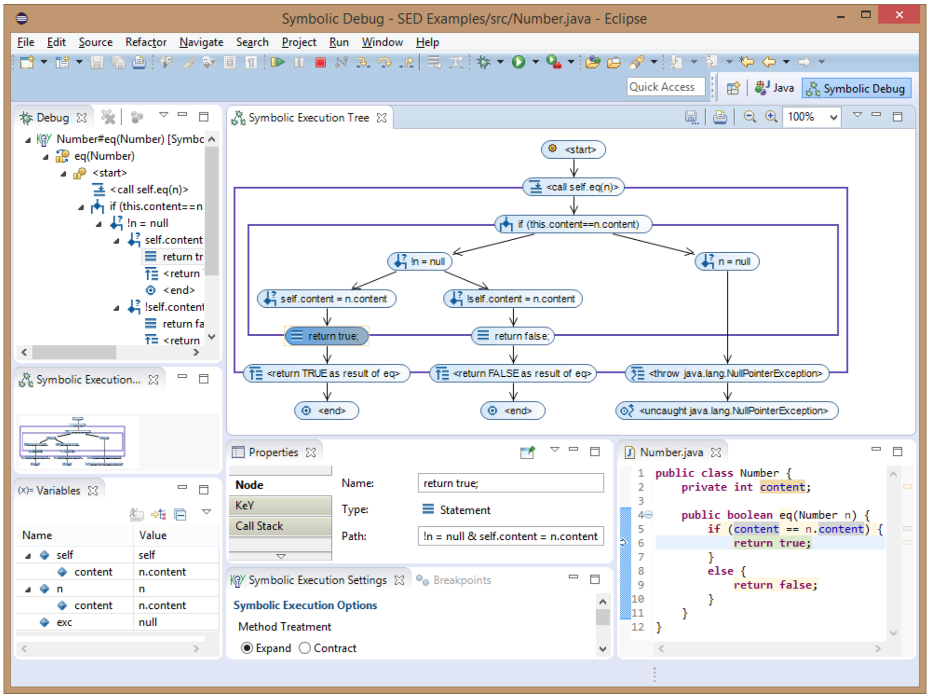
Fig. 1: Source code of class Number and SE tree of method eq

it). SE may not terminate in presence of loops and recursive methods which can be avoided by applying loop invariants or method contracts, see Section 4.

3 Basic Usage of the Symbolic Execution Debugger

The SED is realized as an Eclipse plugin. SE of a selected method or selected statements in a method can be started via the Eclipse context menu item *Debug As, Symbolic Execution Debugger (SED)*. The user is then offered to switch to the *Symbolic Debug* perspective, which provides all relevant views for interactive symbolic execution (see Fig. 2).

The *Debug* view allows, as usual, to switch between debug sessions and to control program execution. Instead of the current stack trace of active threads, the view shows the traversed SE tree. An alternative and more sophisticated visualization of the SE tree is shown in the *Symbolic Execution Tree* view. To ease navigation within large SE trees a thumbnail view called *Symbolic Execution Tree (Thumbnail)* is provided. The SE tree of the screenshot (Fig. 2) is identical to the tree in Fig. 1. The additional frames (blue rectangles) displayed in view *Symbolic Execution Tree* represent the bounds of code blocks. Such frames can be independently collapsed and expanded to abstract away from the inner structure of code blocks, thus achieving a cleaner representation of the overall code structure by providing only as much detail as required for the task at hand. A collapsed frame contains only one branch condition node per path (namely the conjunction of all branch condition of that particular path), displaying the constraint under which the end of the corresponding code block is reached.



Name	Value
self	self
content	n.content
n	n
content	n.content
exc	null

Name: return true;

Type: Statement

Path: !n = null & self.content = n.content

Method Treatment

● Expand ○ Contract

```
public class Number {
    private int content;

    public boolean eq(Number n) {
        if (content == n.content) {
            return true;
        }
        else {
            return false;
        }
    }
}
```

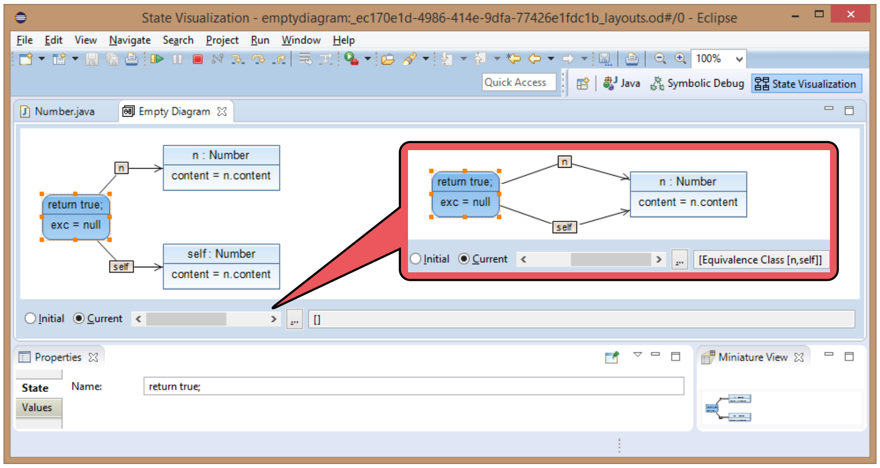
Fig. 2: Symbolic Execution Debugger: interactive symbolic execution

The symbolic program state of a node consists of variables and their symbolic values. It can be inspected in the *Variables* view. The details of a selected variable (e.g. additional constraints) or node (path condition, call stack, etc.) are available in the *Properties* view. The source code line corresponding to the selected SE tree node is highlighted in the editor. Additionally, the editor highlights statements and code members reached during symbolic execution.

The *Symbolic Execution Settings* view lets one customize SE, e.g., one can choose between method inlining and method contract application. Breakpoints suspend the execution and are managed in the *Breakpoints* view.

In Fig. 2 the SE tree node **`return true`**; is selected. In the *Variables* view we can see that the symbolic values of field `value` are identical for the objects referenced by `self` (the current instance) and parameter `n`. This is exactly what is enforced by the path condition. A fallacy and source of bugs is to implicitly assume that `self` and `n` refer to different instances as they are named differently and here also because that an object is passed to itself as a method argument. But the path condition is also satisfied if `n` and `self` reference the same object. The SED helps to detect and locate unintended aliasing by determining and visualizing all possible memory layouts w.r.t. the current path condition.

Selecting context menu item *Visualize Memory Layouts* of an SE tree node creates a visualization of possible memory layouts as a *symbolic object diagram*



Name: return true;

Fig. 3: Symbolic Execution Debugger: different memory layouts

(see Fig. 3). It resembles a UML object diagram and shows the dependencies between objects, the values of object fields and the local variables of the current state.

The root of the symbolic object diagram is visualized as a rounded rectangle and shows all local variables visible at the current node. In Fig. 3, the local variables `n` and `self` refer to objects visualized as rectangles. The content of the instance field `value` is shown in the lower compartment of each object. The local variable `exc` is used by KeY to distinguish among normal and exceptional termination.

The toolbar (near the origin of the callout) allows to select different possible layouts and to switch between the current and the initial state of each layout. The initial state shows how the memory layout looked before the execution started resulting in the current state. Fig. 3 shows both possible layouts of the selected node **`return true`**`;` in the current state. The second memory layout (inside the callout) represents the situation, where `n` and `self` are aliased.

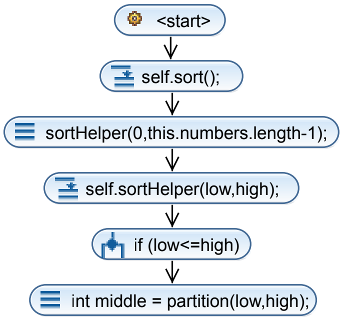
4 Usage Scenarios

Like a traditional debugger, the SED helps the user to control execution and to comprehend each performed step. It is helpful to focus on a single branch where a buggy state is suspected. (To change the focus to a different branch, no new debugging session or new input values are needed). It is always possible to revisit previous steps, because each node in the SE tree provides the full state.

Finding the Origin of Bugs The explicit rendering of different control flow branches in the SE tree constitutes a major advantage over traditional debuggers. Unexpected or missing expected branches are good candidates for possible sources of bugs. Fig. 4a shows a buggy part of a Quicksort implementation for sorting array `numbers`. Within a concrete execution of a large application a `StackOverflowError` was thrown. It indicates that method `sortHelper` calls itself infinitely often. Using SED we start debugging close to the suspected location of the bug, namely, at method `sort`. Executing the method stepwise, exhibits execution paths taken when invoking the method in an illegal state. Exploration of such cases can be avoided by providing a precondition which limits the initial symbolic state. In this example, we exclude empty arrays by specifying the precondition `numbers != null && numbers.length >= 1` in the *debug configuration*. After a few steps, the SE tree produced by SED (see Fig. 4b) shows that the **if** statement is not branching. This is suspicious and deserves closer attention. Inspecting the **if** guard shows that the comparison should have been `low < high` and the source of the bug is found.[3]

```
public class QuickSort {
   private int[] numbers;

   public void sort() {
      sortHelper(0, numbers.length - 1);
   }

   private void sortHelper(int low, int high) {
      if (low <= high) {
         int middle = partition(low, high);
         sortHelper(low, middle);
         sortHelper(middle + 1, high);
      }
   }

   private int partition(int low,
                         int high) {
      // ...
   }
}
```

(a) Buggy Quicksort implementation (from [6])


<start>
self.sort();
sortHelper(0,this.numbers.length-1);
self.sortHelper(low,high);
if (low<=high)
int middle = partition(low,high);

(b) SE tree

Fig. 4: Quicksort example

Program and Specification Understanding SE trees show control and data flow at the same time. Thus they can be used to help understanding programs and specifications just by inspecting them. This can be useful during code reviews or in early prototyping phases, where the full implementation is not yet available. It works best, when partial method contracts and invariants are available to achieve

[3] Without the precondition the bug can be observed as well, but a little later.

compact and finite SE trees. However, useful specifications can be much weaker than what would be required for verification. The listing in Fig. 5 shows a buggy implementation of method `indexOf` with a very simple loop invariant written in the Java Modeling Language (JML) [12]. We configured the symbolic execution engine to apply loop invariants instead of unrolling loops, which guarantees a finite SE tree. The resulting SE tree under precondition `a != null` is also shown in Fig. 5. Application of the loop invariant splits execution into two branches. *Body Preserves Invariant* represents all loop iterations and *Use Case* continues execution after the loop (full branch conditions are not shown for brevity).

Even without checking any further details, it is already indicated by the icon crossed out in red that the leftmost branch terminates in a state where the loop invariant is not preserved. Now, closer inspection shows the reason to be that, when the array element is found, the variable `i` is not increased, hence the **decreasing** clause (`a.length - i`) of the invariant is violated. The two branches below the *Use Case* branch correspond to the code after the loop has terminated. In one case an element was found, in the other not. Looking at the return node, however, we find that in both cases instead of the `index` computed in the loop, the value of `i` is returned.

```java
public static int indexOf(int[] a,
                          int   s) {
   int index = -1;
   int i = 0;
   /*@ loop_invariant i >= 0 && i <= a.length;
     @ decreasing a.length - i;
     @ assignable index, i;
     @*/
   while (index < 0 && i < a.length) {
      if (s == a[i]) { index = i; }
      else { i++; }
   }
   return i;
}
```

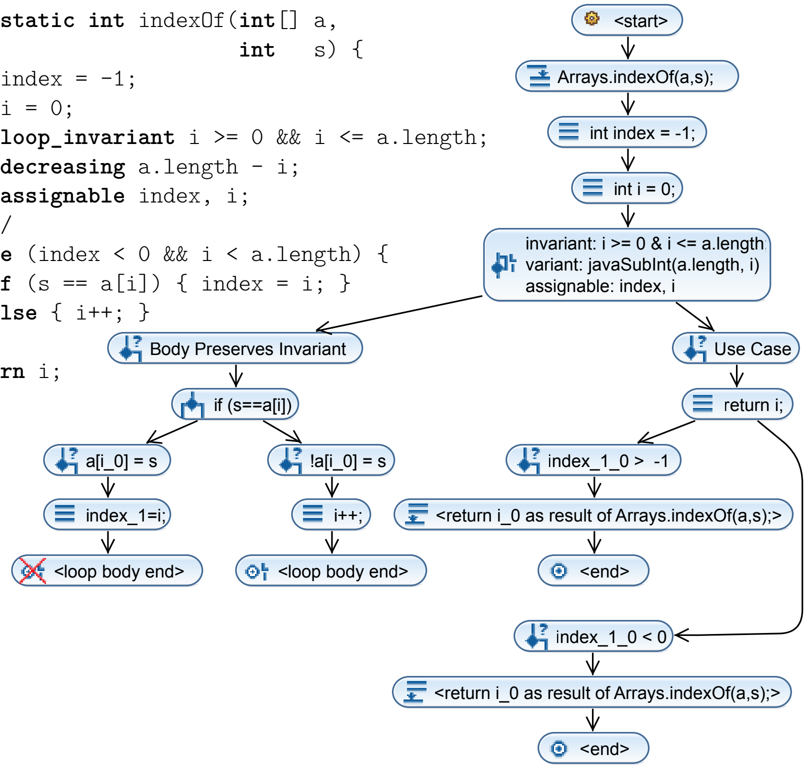


Fig. 5: Buggy and partially specified implementation of `indexOf` and its SE Tree

Our examples demonstrate that SE trees can be used to answer questions about thrown exceptions or returned values. In SED the full state of each node is available and can be visualized. Thus it is easily possible to see whether and where new objects are created and which fields are changed when (comparison between initial and current memory layout).

Using breakpoints, symbolic execution is continued until a breakpoint is hit on any branch. Breakpoints can be attached to a line of code with or without a condition or they may consist only of a condition. Thus they can be used to find execution paths that (i) throw a specified exception, (ii) access or modify a specified field, (iii) invoke or return from a specified method. Breakpoints can also be used to (iv) control loop unwinding and recursive method invocation and (v) to stop at an intermediate state that has a specified property.

5 Verification with SED and KeY

The SED platform allows to perform SE interactively, to visualize the resulting symbolic execution tree, and to inspect symbolic states. Together, this results in a powerful debugging tool that in addition can be used to control SE and to present results of an SE-based analysis.

Going beyond mere SE, the SED can also verify that a Java program satisfies a given specification written in JML, because it uses KeY as its underlying symbolic execution engine.[4] A program is correct with respect to its specification if and only if each branch in the SE tree ends with a termination node and no icons are crossed out in red are displayed in the whole tree. In this case all branches terminate in a state where the given postcondition (i.e., JML ensures clause) is fulfilled. If a method call was approximated by a method contract, the precondition- and caller-no-null checks must have been successful, too. In addition, all loop invariants present were valid at the start of their loop and were preserved by the loop body.

An SE tree produced by the SED displays considerably less information than a full proof tree in KeY [2]: while the former contains only nodes that correspond to reachable program states, the latter shows all intermediate SE steps performed during proof construction, including the proof steps for pure first-order verification conditions. Hence, the SED provides a software developer's view on a KeY proof, hiding intermediate and non-SE related steps. Program states are visualized in a user-friendly way and are not encoded as formulas often distributed and hidden within large proof goals. A major limitation of the SED compared to the KeY prover is that it is currently not possible in SED to continue a proof interactively in case KeY's proof strategy was not powerful enough to close some goal automatically. But it provides still the means to interact with the prover by adapting or inserting additional JML assertions and thus to use KeY with an auto-active flavor [16,13].

[4] The debug configuration allows to select a method contract alternatively to a precondition.

6 Architecture

The SED extends Eclipse and can be added to existing Eclipse-based products. In particular, SED is compatible with Eclipse's Java Development Tools (JDT). To ensure compatibility and to obtain a seamless integration with the Eclipse user interface, SED uses and extends the Eclipse platform as shown in Fig. 6.

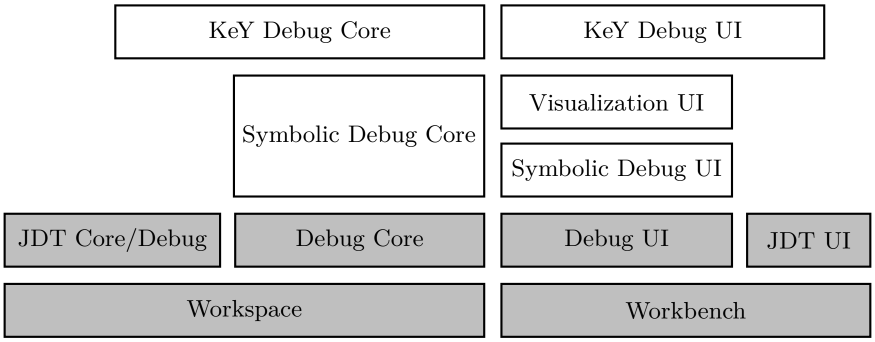


Fig. 6: Architecture of the Symbolic Execution Debugger (SED). Eclipse components are shaded in grey, our extensions have a white background.

The core of Eclipse is the Workspace which manages the projects and the user interface (Workbench) with its editors, views and perspectives. The Debug Platform extends these with language independent facilities for debugging (Debug Core and Debug UI). Finally, JDT offers functionality to edit and debug Java programs (JDT Core/Debug and JDT UI).

The Symbolic Debug Core component extends the debug model of the Debug Platform for symbolic execution, independently from specific target languages and symbolic execution engines. Additional UI extensions (Symbolic Debug UI) and visualization capabilities (Visualization UI) are available.

The KeY Debug Core component implements the extended debug model for symbolic execution based on KeY's symbolic execution engine (available as pure Java API). The user interface extensions required to launch Java methods and to execute statements symbolically are provided by the KeY Debug UI.

The architecture of the SED platform allows us to integrate different symbolic execution engines for the purpose of debugging, program understanding, and to control analyses based on symbolic execution. In order to integrate a new SE engine, it suffices to implement the extended debug model for symbolic execution and to create the user interface extensions to start and control the symbolic execution. [5]

[5] The website www.key-project.org/eclipse/SED provides additional documentation and an example SED implementation as a starting point.

7 Related and Future Work

A number of recent tools implement SE for program verification [9] or test generation [1,15], which are complementary to SED. In fact, SED could be employed to control or visualize these tools. As far as we know, EFFIGY [11] was the first system that allowed to interactively execute a program symbolically in the context of debugging. It did not support specifications or visualization.

The Eclipse plugin of Java Path Finder (JPF) [14] prints the analysis results obtained from SE as a text report, but does neither provide graphical visualization nor interactive control of SE. JPF is prototypically supported by SED as an alternative SE engine.

The SE engine and its Eclipse integration described in [8] features non-interactive graphic visualization of the SE tree. SED allows to interact with the visualization as a means to control SE and to inspect symbolic states.

A prototypic symbolic state debugger that could not make use of method contracts and loop invariants was presented in [6]. However, that tool was not very stable and its architecture was tightly integrated into the KeY system. As a consequence, the SED was developed from scratch as a completely new application featuring significant extended and new functionality. It is realized as a reusable Eclipse extension which allows to integrate different SE engines.

We plan to extend the verification capabilities of the SED to create a full-fledged alternative GUI of the KeY verification system [2]. The visualization capabilities and a debugger-like interface will flatten the learning curve to use a verification system. On the other hand, exploiting verification results during SE allows to classify execution paths automatically as correct or wrong.

References

1. E. Albert, I. Cabanas, A. Flores-Montoya, M. Gomez-Zamalloa, and S. Gutierrez. jPET: An Automatic Test-Case Generator for Java. In *Proc. of the 18th Working Conf. on Reverse Engineering*, WCRE '11, pages 441–442. IEEE CS, 2011.
2. B. Beckert, R. Hähnle, and P. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2007.
3. R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT—A formal system for testing and debugging programs by symbolic execution. *ACM SIGPLAN Notices*, 10(6):234–245, June 1975.
4. R. M. Burstall. Program proving as hand simulation with a little induction. In *Information Processing '74*, pages 308–312. Elsevier/North-Holland, 1974.
5. M. Grottke and K. S. Trivedi. A classification of software faults. *Journal of Reliability Engineering Association of Japan*, 27(7):425–438, 2005.
6. R. Hähnle, M. Baum, R. Bubel, and M. Rothe. A visual interactive debugger based on symbolic execution. In *ASE*, pages 143–146, 2010.
7. M. Hentschel, R. Bubel, and R. Hähnle. Symbolic Execution Debugger (SED). In B. Bonakdarpour and S. A. Smolka, editors, *Proceedings of Runtime Verification 2014*, LNCS, pages 255–262. Springer, Sept. 2014.
8. A. Ibing. Parallel SMT-Constrained Symbolic Execution for Eclipse CDT/Codan. In *ICTSS*, pages 196–206, 2013.

9. B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *Proc. of the 3rd Intl. Conf. on NASA Formal Methods*, pages 41–55. Springer, 2011.
10. S. Katz and Z. Manna. Towards automatic debugging of programs. In *Proc. of the Intl. Conf. on Reliable software, Los Angeles*, pages 143–155. ACM Press, 1975.
11. J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.
12. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, and D. M. Zimmerman. *JML Reference Manual*, Sept. 2009.
13. K. R. M. Leino and M. Moskal. Usable Auto-Active Verification. `http://fm.csl.sri.com/UV10/submissions/uv2010_submission_20.pdf`, 2010.
14. C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing Nasa Software. In *Proc. of the 2008 Intl. Symposium on Software Testing and Analysis*, ISSTA '08, pages 15–26. ACM, 2008.
15. N. Tillmann and J. De Halleux. Pex: White Box Test Generation for .NET. In *Proc. of the 2nd Intl. Conf. on Tests and Proofs*, pages 134–153. Springer, 2008.
16. J. Tschannen, C. A. Furia, M. Nordio, and N. Polikarpova. Autoproof: Auto-active functional verification of object-oriented programs. In C. Baier and C. Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 9035 of *Lecture Notes in Computer Science*, pages 566–580. Springer, 2015.

The platin Tool Kit - The T-CREST Approach for Compiler and WCET Integration

Stefan Hepp[1], Benedikt Huber[2], Jens Knoop[1], Daniel Prokesch[2], and Peter Puschner[2]

[1] Institute of Computer Languages
Vienna University of Technology
Vienna, Austria
{hepp,knoop}@complang.tuwien.ac.at

[2] Institute of Computer Engineering
Vienna University of Technology
Vienna, Austria
{benedikt,daniel,peter}@vmars.tuwien.ac.at

Abstract. The construction of safety-critical real-time applications requires predictable computer platforms that enable a safe and tight static analysis of those systems. The worst-case performance and the availability of tight bounds on the worst-case execution time (WCET) of the tasks of the applications are of central importance in such systems.

The compiler tool-chain plays an integral part in a real-time platform infrastructure. Not only is the compiler responsible for relating source code level annotations such as flow facts to the generated machine code – a task necessary to achieve high-quality bounds on optimised code. Also, the analysis tools profit from program information that is readily available in the compiler but difficult to retrieve from the generated binary alone. Therefore, the compiler should export internal knowledge about the program to the analysis. Furthermore, compilation for hard real-time systems requires optimisations that specifically aim at reducing the worst-case execution path instead of reducing the average case performance. This requires feedback from the worst-case analysis back to the compiler.

In this paper we describe the our approach to the compiler tool integration that has been realised in the platin tool kit, developed in the EU FP7 T-CREST project. The platin tool kit is a portable glue tool that interfaces our LLVM-based T-CREST compiler with several research and industrial strength analysis tools. Our approach is transferable to other compiler tool-chains and minimises the effort for adapting them for the requirements of real-time platforms.

1 Introduction

Embedded computer systems are playing an increasingly important role in applications that are time-critical, e.g., in fly-by-wire applications, in medical equipment, and in control systems of nuclear power plants. To ensure safety, the computer systems controlling the actuators in these applications have to respond to changes in the environment within strict time bounds. It is thus important to design and implement these systems to meet their timing constraints and to show that the implementation indeed fulfils all timing requirements. Despite the stringent timing requirements of these time-critical applications, the importance of *time as a first-order property of embedded systems behaviour* is not adequately reflected by the platforms and methods/tools widely used for the construction of the embedded computer systems for these applications.

Within the T-Crest project a new embedded multi-core platform was developed [14], which emphasised time-predictability in all design decisions. The T-Crest platform consists of a novel processor core called Patmos [13], a time-predictable network-on-chip (NoC) that connects the cores to each other and to a predictable memory controller, and a tool chain centred around LLVM [9] for compiling and analysing applications written in C code [11].

The primary task of the compiler tool chain in such a platform is to generate machine code for the target architecture. However, the compiler should also to try to minimise the worst-case execution time (WCET) of the application tasks, and support the worst-case analysis tools in finding tight and safe WCET bounds. This requires interaction of the compiler with the WCET analysis tools. In the T-Crest project we implemented common routines for tool integration and analysis in a separate tool kit called *platin* that requires only small adoptions of the compiler and can be reused for other target architectures as well. The platin tool kit is centred around its native *PML* file format that stores information about the program structure and meta-information such as flow facts and analysis results in a target-machine agnostic form. The tool kit not only contains tools to interface with external analysis tools such as the industry-standard AbsInt aiT WCET analyser, it also provides tools for tasks such as flow fact transformation, WCET analysis, graph visualisation and tool configuration.

In this paper we give an overview of the T-Crest platform, its compiler tool chain and the platin tool kit. We present how the platin tool kit binds the compiler and the WCET analysis tools together, show how to use use the platin tool kit, and briefly discuss the steps required to adapt platin for a new target architecture. The rest of the paper is structured as follows. Section 2 introduces the T-Crest platform, while Section 3 overviews the Patmos compiler tool chain. Section 4 presents the platin tool kit its interaction with the compiler and analysis tools, and overviews the tools provided by platin. Section 5 demonstrates the use of platin by means of an example. We discuss related work in Section 6 and conclude this paper with Section 7.

2 The T-CREST Platform

The goal of the T-Crest project[3] was to develop a fully time-predictable multi-core platform. The T-Crest platform consists not only of a novel processor

[3] Results and publications of the T-Crest project are available from the project website `http://www.t-crest.org/`

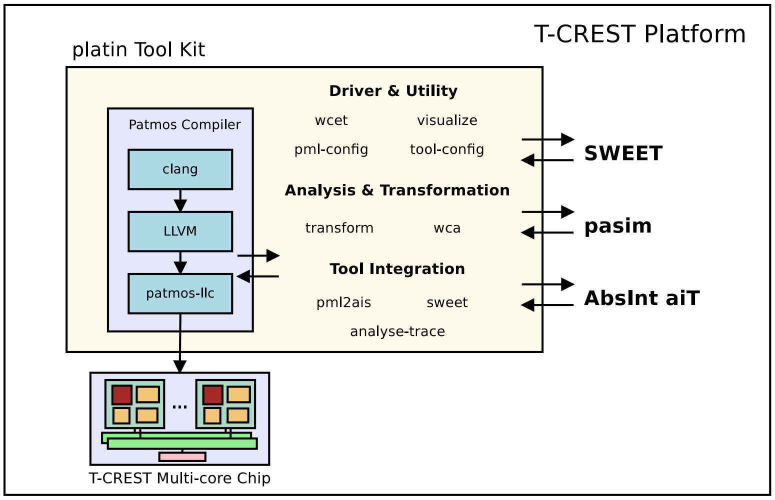


Fig. 1: The T-CREST platform and its compiler and analysis software ecosystem.

core, but also includes a time-predictable memory system, a compiler, analysis tools and runtime libraries. All of them are designed to play together to achieve a highly predictable platform for embedded systems.

The hardware side of the T-Crest platform consists of the T-Crest multicore *chip*. It contains a configurable number of *processor cores*. Each core contains a Patmos *processor* and several local memories. The Patmos processor [13] uses a fully-predicated 32-bit RISC-style instruction-set architecture (ISA) and a five-stage in-order pipeline. Each core features a data cache, a stack cache [1], a method cache [2] and a local scratchpad. The cores are connected to each other by a time-predictable network-on-chip [7] called Argo, which can be used for message passing. A separate time-predictable memory interconnect [4] called Bluetree connects the cores to the memory controller [5] for the shared RAM.

Figure 1 gives a software-centric overview of the T-CREST platform. The LLVM-based Patmos *compiler tool chain* uses the clang C frontend to parse C code into LLVM bitcode, which is then optimised by LLVM bitcode passes. The LLVM backend for Patmos `patmos-llc` generates machine code for the Patmos processor. The platin tool kit [11] is a key component in the T-Crest platform for tool integration and WCET analysis. It is tightly coupled with the compiler tool chain and serves three main tasks. First, it provides tools for flow fact transformation and for program analysis. Second, it interfaces with existing analysis tools in order to bring their analysis functionality to the T-Crest platform. Among the supported tools are the SWEET flow analyser and the industry-standard AbsInt aiT WCET analyser. Platin can also be used to analyse execution traces generated by the Patmos simulator `pasim`. Third, platin provides utility tools for result visualisation and tool configuration, as well as driver tools that chain multiple analysis and transformation steps into single, easy to use commands.

We will overview the Patmos compiler tool chain in the next section, while the rest of this paper presents the platin tool kit in more detail.

3 The Patmos Compiler

The Patmos processor developed within T-Crest is designed for high time predictability [13]. The architectural features of this processor are designed to improve performance yet remain inherently timing analysable. This is achieved by using static (compile-time) alternatives for commonly used performance-enhancing features at runtime in order to reduce hard-to-analyse dynamic behaviour. A worst-case timing analysis tool can then be used to derive tight WCET bounds for the real-time tasks of the embedded application.

The task of the Patmos compiler is thus twofold. First, the compiler must generate code that targets the Patmos ISA and exerts control over the components of the processor core so that the generated program exhibits a low WCET [2,1]. Second, the compiler must support the WCET analysis by providing information available in the compiler that usually is discarded but is valuable for automated and precise timing analysis. This includes preserving information about the control-flow structure, but also flow annotations provided by the user that constrain the possible flow of control, e.g., bounds on the maximum number of loop iterations (*loop bounds*). The compiler in turn can profit from static analysis results from the timing analysis to guide optimisations towards a good worst-case performance. This requires an integration of the compiler and the WCET analysis tools.

Figure 2 gives an overview of the compiler tool chain. The compiler is based on the LLVM compiler framework [9]. At the beginning of the compilation process, each C source code file is translated to LLVM intermediate representation (*bitcode*) by the C frontend `clang`. The user application code as well as standard C libraries and runtime support libraries are linked on this intermediate level by the `llvm-link` tool, presenting subsequent analysis and optimisation passes as well as the code generation backend a complete view of the whole program. This control-flow graph (CFG) oriented intermediate representation is particularly suitable for generic target independent optimisations, such as common subexpression elimination, which are readily available through the LLVM `opt` tool. The `llc` tool constitutes the compiler backend. It translates LLVM bitcode into machine code for the Patmos ISA, addressing the target-specific features for time predictability. The backend produces a relocatable ELF binary containing symbolic address information, which is processed by `gold`,[4] defining the final data and memory layout, and resolving symbol relocations. An important property of this compilation flow stems from the fact that the application is already linked at intermediate level: Optimisations and the code generator have a complete

[4] gold is part of the GNU binutils, see `http://sourceware.org/binutils/`

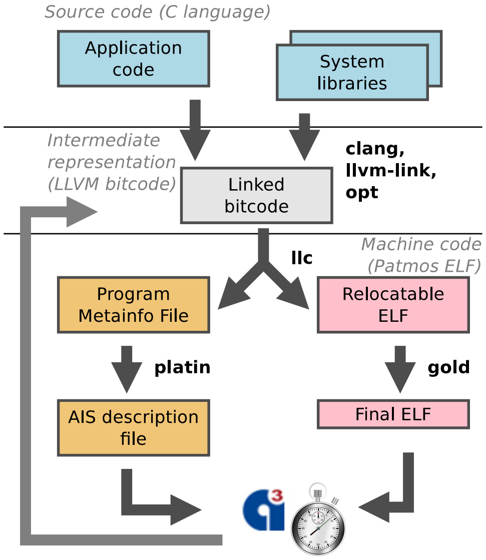


Fig. 2: Overview of the Patmos compiler tool chain

view of the program, which is necessary for optimisations that need to balance the use of a shared resource across the whole program execution. For example, Patmos' specialised software-controlled caches require the compiler to be aware of all cache accesses along the worst-case path for it to be able to generate code that exhibits lowest possible WCET.

In addition to the machine code, the backend exports complete information about the control-flow structure of both bitcode and machine code as well as information about the program obtained by the compiler in the Program Metainfo Language (PML) format, as detailed in the following section. The `platin` tool kit uses these PML files to perform analysis tasks and to transform flow facts. It is also able to export program information to analysis tools such as the AbsInt WCET analysis tool `aiT`. Analysis results are imported back into the PML file, which can in turn be passed back to the compiler for iterative WCET driven optimisation.

The platform, including the processor, a simulator, the compiler tool chain including the platin tool kit as well as a set of benchmarks is available as open-source from the T-CREST organisation at github.[5] The Patmos handbook [12] provides detailed information about the installation, a description of the processor core and its instruction set architecture (ISA) and documents the use of the compiler tool chain.

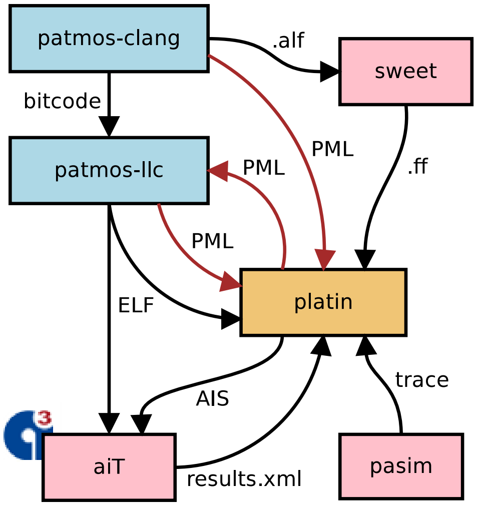


Fig. 3: The platin tool interacts with the compiler with its native PML file, while it communicates with other tools using exporters and importers for their file formats.

4 The Platin Tool Kit

The main task of the platin tool kit is the tool integration in the T-Crest platform. It uses a YAML[6] based file format called PML as its native file format, which stores control flow information, flow facts and value facts, analysis results, information to relate different code representations and a hardware description. Apart from performing tool integration tasks, the platin tool kit has been extended to provide tools not only for visualisation and inspection of information stored in PML files but also to include its own set of cache and path analyses to perform a WCET analysis. Section 4.2 overviews platin's main tools.

Platin's interaction with other tools is shown in Figure 3. It gathers information from a number of sources. The LLVM backend provides the control-flow and call targets on both bitcode and machine-code level. The LLVM PML exporter also retrieves value facts about data pointers as well as flow facts from LLVM-internal analysis passes and adds them to the generated PML file. The platin tool kit also contains several analysis drivers for external analysis tools. The trace analysis tool derives execution timings and flow facts from a simulator's execution trace (`pasim` for Patmos). Platin can also use the abstract interpretation based analyser SWEET [10] to find additional flow facts. An LLVM plugin exports the LLVM bitcode representation of the program to the Artist

[5] `http://www.github.com/t-crest`

[6] A "human friendly data serialization standard for all programming languages", see `http://yaml.org/`

Flow Analysis Language (ALF) that SWEET uses as input language. Flow facts found by SWEET are then imported by platin and mapped back to bitcode.

Using the information provided by the compiler, platin is able to transform the gathered information from bitcode level to the machine code level. The combined set of flow facts and value facts is then passed on the AbsInt WCET analyser aiT in aiT's native AIS format, or used in the internal WCET analysis tool called WCA. A flow fact simplification step ensures that the flow facts are expressed in a form that is understood by the used WCET analysis tool.

4.1 Flow Fact Transformation

LLVM splits the task of compiling source code to machine code into two major steps: compiling to and optimising on a machine-independent intermediate representation called bitcode (`clang`) and lowering bitcode down to machine code using machine specific optimisations in the backend (`llc`). The LLVM backend does not change the control flow in a major way since optimisations such as loop transformations and inlining are performed at bitcode level. This enables platin to map bitcode and machine code across the backend automatically using relation graphs [6] with almost no adaption of the backend. Using these relation graphs, platin is able to transform linear flow facts from bitcode to machine code without further user assistance.

Maintaining flow facts across high-level optimisations is inherently more difficult and requires at least some compiler support. There are various approaches to that problem. Transforming the flow facts along with the optimisation transformations can be done either by the compiler itself as implemented in the WCC compiler [3], or by an external tool that requires a log of all compiler transformations as proposed by Kirner et al˙ [8]. Platin leaves the task of high-level flow fact transformation to the compiler. The Patmos compiler must therefore ensure that flow facts that are exported to PML match the exported optimised bitcode. Flow facts that are derived directly from LLVM analyses do not need to be co-transformed since the LLVM framework itself either updates or reruns analyses after optimisation passes as required. For manual source code annotation, the Patmos compiler currently supports constant loop bound flow facts as source code pragmas. The compiler disables transformations that might invalidate these loop bounds for functions containing such source annotations. The preserved source code pragmas can thus be directly exported to PML.

For the future we plan to support arbitrary linear flow facts in the Patmos compiler by using source code markers. In contrast to other techniques, using flow markers requires only minimal changes to the compiler. In particular, optimisations do not need to update flow facts as long as the code transformations preserve the sequence of markers on any program path. This makes integrating and maintaining flow fact support in a large existing and constantly evolving compiler such as LLVM much more feasible. Support for source code markers is still under development in the Patmos compiler though.

4.2 Platin Tools

The core of platin is a Ruby framework for working with PML data. It provides common functionality such as reading and writing PML files, accessing and traversing PML data structures, merging and modifying PML data, constructing various graph representations and working with context-sensitive information. The tools and analyses in platin are built on top of that framework. The tools typically accept one or more PML input files and a number of options, and will generate a new PML output file. The tools can be chained together by passing the output PML file of any tool as input of another platin tool. Platin tools can also invoke other platin tools internally in order to implement complex functionality. In this case, PML data is passed between the tools in-memory. The Ruby scripting language enables rapid development of tools and analyses and allows the developer to focus on the task at hand, which is especially essential in a research environment. While a Ruby implementation implies some performance drawbacks compared to other languages, we did not find the performance of the platin tools to be an issue in our experiments.

Platin provides several tools to work with its native PML file format. The platin `pml` tool can merge and validate PML files or print out flow facts, value facts and timing analysis results in a condensed form. The `visualize` tool can be used to visualise control flow graphs and relation graphs.

Platin can also be used for for tool configuration. It uses PML files to configure parameters of the hardware model, such as cache parameters and memory latencies. The platin `pml-config` tool can be used to generate or modify such a PML hardware model, while the `tool-config` tool generates command line options for tools like the compiler and the simulator to configure them consistent with the hardware model. Other tools like the WCET analysis tool and the aiT export also use the PML hardware model configuration to setup the timing parameters.

For tool integration, platin provides tools such as `sweet`, `analyze-trace` and `pml2ais`. The `sweet` tool invokes SWEET to find flow facts. The results are parsed and added to the PML file. The `analyze-trace` tool generates flow facts from simulation runs, which are only valid for the inputs used in the simulation but are useful for testing the correctness and precision of WCET analyses. Flow facts are attached either at bitcode or at machine code level, depending on their source. The `transform` tool converts flow facts between different levels. The `pml2ais` tool in turn exports flow facts to the AbsInt aiT AIS file format and generates an analysis project file for aiT based on the platin configuration.

The platin `wcet` tool is a driver tool for the WCET analysis tools. It invokes either AbsInt aiT using the `pml2ais` exporter or platin's internal WCET analysis `wca`. The `wcet` tool can optionally use many of the above tools to find and transform additional flow facts. It also sets up the analysis tools according to the PML hardware model and provides options to configure specific analysis modes such as always-hit or always-miss cache analyses.

4.3 Integrating platin Into Other Compiler Tool Chains

The platin tool kit has been designed to support multiple architectures with a minimal effort for adapting platin and the compiler tool chain. The PML file format and most of the functionality of platin is architecture independent. Memory latencies and caches are configured in a generic hardware model. Support for analysis tools like aiT and SWEET that support multiple target platforms is implemented in generic platin tools. Architecture dependent analysis and tool integration code is encapsulated in architecture modules in platin. Adapting platin to a new architecture thus only requires the implementation of a new architecture module for that platform, which invokes platform-specific tools such as a simulator and performs basic analysis tasks such as deriving the WCET of a basic block.

Platin requires a compiler backend that generates PML files. For LLVM backends, the PML export machine-function pass can be reused, as it also has been implemented in a generic way. This is possible due to the generic representation of machine code in LLVM backends. The PML export pass creates PML files, exports the structure of machine code, bitcode and relation graphs. Only the classes that retrieve target-specific information such as call or jump targets and interface with backend analysis passes need to be specialised. Work on high-level support for flow-fact transformation in the `clang` frontend and on bitcode level can be reused directly from the Patmos compiler, since the frontend and middle-end is platform independent.

Platin fully supports the Patmos platform and has some initial support for an ARM tool chain support. We do believe that basic support for other LLVM based compiler tool chains can be achieved comparatively quickly, as only a few key components in the LLVM backend and in platin need to be implemented or adapted. As a result and due to platin's open source nature, the platin tool kit can be useful for other projects in the domain of embedded real-time systems as well.

5 Example

In this section we demonstrate some of the tools of `platin`. We show a typical workflow by compiling and analysing a small demo application on Patmos. A quick start guide for installing the Patmos tool chain can be found in the Readme file of the Patmos repository[7] or in the Patmos handbook [12].

Listing 1 shows the content of `sort.c`. It contains a simple insertion sort implementation in function `sort`. Our target function for analysis is `gen_sort`, which fills an array with N pseudo-random numbers and then sorts the array. In order to prevent the compiler from inlining and removing our analysis target function, we mark the function as `noinline`. The code contains loop bound annotations for the WCET analysis in the form of pragmas.

[7] `https://github.com/t-crest/patmos`

Listing 1: Demo application that initialises and sorts an array.

```c
#include <stdlib.h>

#define MAX_SIZE 100

void sort(int *arr, size_t N) {
  #pragma loopbound min 0 max 99
  for (int j = 1; j < N; j++) {
    int i = j - 1;
    int v = arr[j];
    #pragma loopbound min 0 max 99
    while (i >= 0 && arr[i] >= v) {
      arr[i+1] = arr[i];
      i = i - 1;
    }
    arr[i+1] = v;
  }
}
void gen_sort(int *arr, size_t N) __attribute__((noinline));
void gen_sort(int *arr, size_t N) {
  #pragma loopbound min 1 max MAX_SIZE
  for (size_t i = 0; i < N; i++) {
    arr[i] = rand() % N;
  }
  sort(arr, N);
}
int main(int argc, char** argv) {
  srand(0);
  int arr[MAX_SIZE];
  size_t N = rand() % (MAX_SIZE / 2) + (MAX_SIZE / 2);

  gen_sort(arr, N);

  return 0;
}
```

All tools in the Patmos tool chain are configured to use the default Patmos hardware configuration if no further options are given. In this example we show how to use `platin` to configure a different hardware setup. For this, we use `pml-config` to generate a modified hardware model:

```
platin pml-config --target patmos-unknown-unknown-elf \
                  -o config.pml -m 2k -M fifo8
```

This command generates a new `config.pml` file containing a description of the default hardware model, except that we use a method cache of only half the size (2 KB size with a tag memory of 8 entries).

In the next step, we compile our program using the `patmos-clang` compiler driver. We also use the `platin tool-config` tool to setup the compiler according to our modified hardware model. `tool-config` can be used in a similar manner to setup `pasim`, the Patmos simulator. We need to explicitly enable optimisations with `-O2`, as the default optimisation level is `-O0`.

```
patmos-clang ‘platin tool-config -i config.pml -t clang‘ \
             -O2 -o sort -mserialize=sort.pml sort.c
```

Listing 2: Analysis report for the `sort` application

```
---
- analysis-entry: gen_sort
  source: trace
  cycles: 49089
- analysis-entry: gen_sort
  source: platin
  cycles: 644867
  cache-max-cycles-instr: 651
  cache-min-hits-instr: 398
  cache-max-misses-instr: 3
  cache-max-cycles-stack: 0
  cache-max-misses-stack: 0
  cache-max-cycles-data: 436779
  cache-min-hits-data: 0
  cache-max-misses-data: 10599
  cache-max-stores-data: 10200
  cache-unknown-address-data: 20799
  cache-max-cycles: 437430
```

The driver calls all commands necessary to compile the source code, link and optimise the bitcode and generate and link the final binary `sort`. The option `-mserialize` causes the compiler to generate the PML file `sort.pml`. It contains a description of the application control flow at bitcode level (after the bitcode optimisations) and of the final machine code. It also contains value facts and flow facts such as loop bounds as found by the compiler as well as our source-code loop annotations, and relation graphs relating the bitcode and machine code control flow graphs.

Now we are ready to analyse our target function. We use the `platin wcet` driver tool to run all necessary commands, including the trace analysis and the platin WCET analysis tool WCA. The driver tool will automatically try to run the AbsInt aiT analysis tool if it is installed.

```
platin wcet -i config.pml --enable-trace-analysis --enable-wca \
            -b sort -e gen_sort -i sort.pml --outdir tmp \
            -o wcet.pml --report report.txt
```

We need to pass the name of the binary file (`-b`) and both the compiler generated PML file and the hardware model PML file (`-i`) to platin. The `-e` option tells platin the name of the analysis target function. The optional `--outdir` option causes platin to keep temporary files and store them in the given directory, mainly the generated project files for the AbsInt analyser tool `a3patmos`. The optional `-o` option stores detailed analysis results such as the found WCET bounds for the target function, execution timings of basic blocks and execution frequencies of blocks on the worst-case path along with the program information from the input files in a PML file for further analysis or for WCET-driven optimisations.

The `--report` option causes platin to store the result summaries of the analyses in `report.txt`. Listing 2 shows the content of that file. In this example the

Listing 3: Flow facts from LLVM and user annotations as reported by `platin`

```
=== flowfacts generated by llvm.bc ===
--- loop-bound ---
#<FlowFact origin=llvm.bc,level=bitcode, in #<Loop: gen_sort/for.cond>:
  ↪ [1 gen_sort/for.cond] less-equal (1 + %N)>
#<FlowFact origin=llvm.bc,level=bitcode, in #<Loop: gen_sort/for.cond.i>:
  ↪ [1 gen_sort/for.cond.i] less-equal (1 umax %N)>
#<FlowFact origin=llvm.bc,level=bitcode, in #<Loop: __umodsi3/for.cond.i>:
  ↪ [1 __umodsi3/for.cond.i] less-equal 33>
#<FlowFact origin=llvm.bc,level=bitcode, in #<Loop: __umodsi3/for.cond.i>:
  ↪ [1 __umodsi3/for.cond.i] less-equal 33>
=== flowfacts generated by user.bc ===
--- loop-bound ---
#<FlowFact origin=user.bc,level=bitcode, in #<Loop: gen_sort/for.cond>:
  ↪ [1 gen_sort/for.cond] less-equal 101>
#<FlowFact origin=user.bc,level=bitcode, in #<Loop: gen_sort/for.cond.i>:
  ↪ [1 gen_sort/for.cond.i] less-equal 100>
#<FlowFact origin=user.bc,level=bitcode, in #<Loop: gen_sort/while.cond.i>:
  ↪ [1 gen_sort/while.cond.i] less-equal 100>
#<FlowFact origin=user.bc,level=bitcode, in #<Loop: __umodsi3/for.cond.i>:
  ↪ [1 __umodsi3/for.cond.i] less-equal 33>
```

platin WCET analysis derives a lower WCET bound than aiT. aiT is able to find better loop bounds and thus finds fewer data cache misses for the sort loop, but it assumes higher costs for instruction cache misses than platin.

Both analyses seem to highly over-approximate the actual WCET when compared to the trace results of the execution. However, while we assume that in the worst case the whole array is used, the actual execution only fills and sorts a fraction of the array. Hence the measured execution time is not a good indicator for the worst-case performance.

The inner loop of the sort function is a triangle loop. Our annotated global loop bound of $(N-1)^2$ is thus about a factor of two too large. For loops with constant bounds, LLVM is capable of detecting such triangle loops and deriving the correct bounds automatically. Our PML export uses the LLVM analysis results to generate additional flow facts. platin provides a tool to print all flow facts in a PML file in a compact form.

```
platin pml -i sort.pml --print-flowfacts
```

Listing 3 shows the output of that command. We find our manual loop annotations in the `user.bc` origin section. Note that LLVM inlined the `sort()` function, therefore our loops are now in function `gen_sort`.[8] The loop bounds are expressed as flow constraints on the loop header blocks.[9] We can also see that LLVM managed to find parametric loop bounds for two loops, but failed to find a loop bound for the inner triangle loop since in our case the size of the

[8] Function `__umodsi3` implements the modulo operator, as Patmos does not provide a modulo instruction in hardware.

[9] The right-hand side of the constraint is larger than our loop bound by one because the loop header is executed one additional time more than the loop body to jump out of the loop when the loop condition becomes false.

array to sort is not fixed but parametric. It is thus necessary to annotate the inner loop manually. Platin supports arbitrary linear flow constraints in PML. It is possible to manually supply additional flow constraints in PML format. Support for source code flow annotations beyond local loop bounds in the Patmos compiler is planned for future development.

We can also use platin to visualise control-flow graphs, call-graphs and relation graphs:

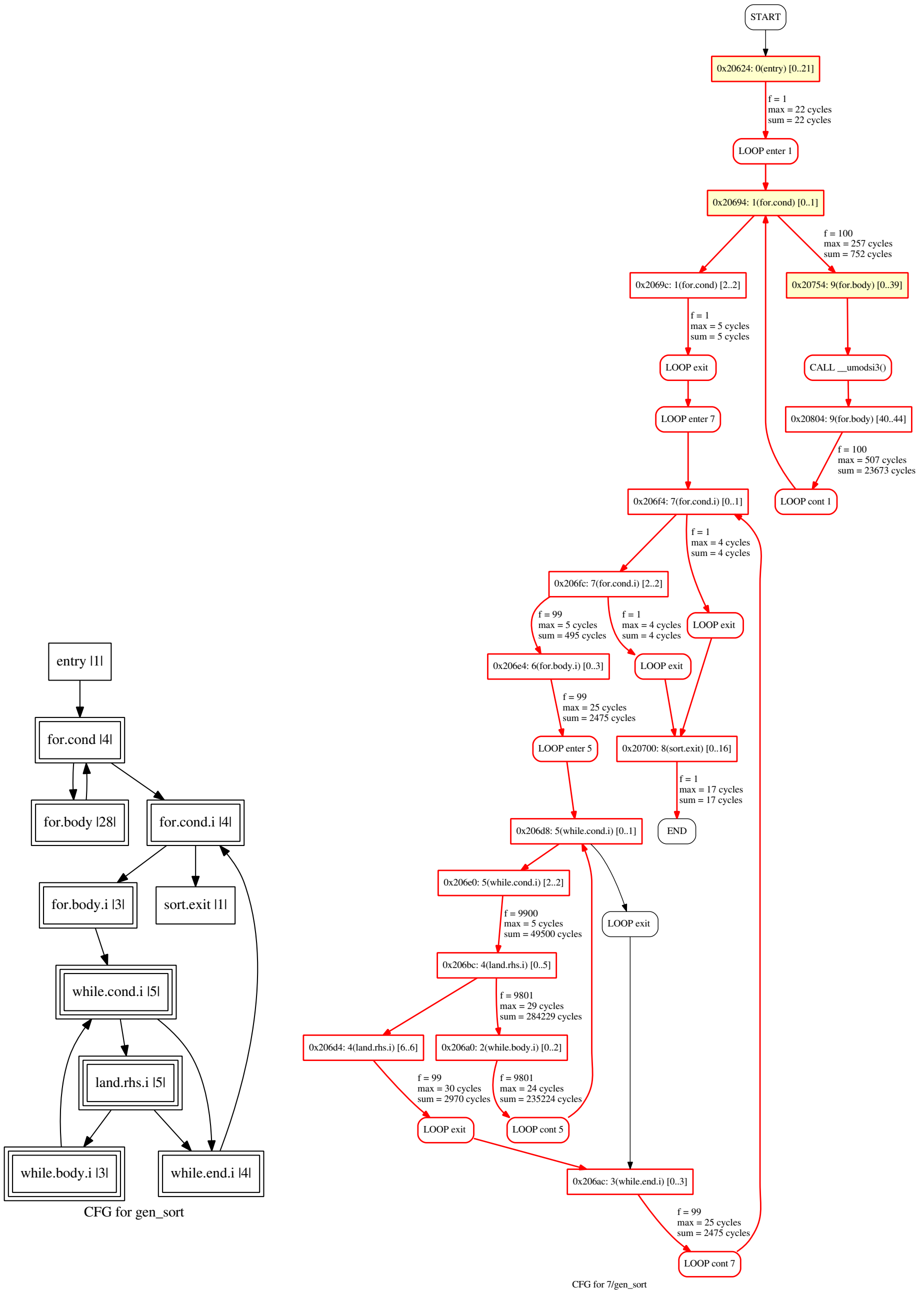
```
platin visualize -i wcet.pml -o out -f gen_sort \
                 --show-timings=platin
```

This command generates all graphs for function `gen_sort` and stores them in the output directory `out`. Figure 4 shows the generated control-flow graphs at bitcode level (after optimisation) and of the final machine code. The latter graph is the same graph that is used for WCET analysis by platin. Square boxes correspond to basic blocks or basic block slices, while round boxes are virtual nodes inserted by platin. The block node labels in the machine code graph show the address and the number of the basic block, as well as the name of the corresponding bitcode block (in brackets) and the range of the instructions in the basic block slice (in square brackets). The `--show-timings` option causes platin to highlight blocks and edges that are on the worst-case path found by the given analysis tool in the machine-code graph. Edges between basic blocks are annotated with their worst-case execution frequency and their associated WCET contribution.

6 Related Work

The WCET-aware C Compiler (WCC) [3] is a custom developed C compiler that focuses on WCET optimisation, targeting Infineon TriCore microcontrollers. It uses a machine-independent high-level intermediate representation called ICD-C for high-level optimisations, and a retargetable low-level intermediate representation called ICD-LLIR for machine optimisations and code generation. WCET analysis is performed by the AbsInt `aiT` tool at ICD-LLIR level and adds analysis results such as basic block execution times and encountered instruction cache misses, as well as information about the found worst-case path to the ICD-LLIR. The compiler maintains a mapping between the blocks of the ICD-C and ICD-LLIR representations, so that WCET analysis results can be used by high-level optimisations on ICD-C as well. Flow facts are transformed and updated by compiler and its optimisation passes itself.

Kirner et al. transform flow information in parallel to high-level optimisations such as loop interchange [8]. Their transformation technique requires control-flow update rules for optimisations that modify the control-flow graph or change loop bounds or other flow constraints. These update rules specify the relation between edge-execution frequencies before and after the optimisation, and are used to consistently transform all flow constraints affected by the optimisation. The method was implemented for source-to-source transformations but should be applicable to bitcode as well.



(a) Bitcode CFG

(b) Machine-code CFG with platin WCET results

Fig. 4: Bitcode and machine-code control-flow graphs for `gen_sort`.

7 Conclusion

In this paper we presented an overview of the Patmos compiler tool chain and the platin tool kit. The platin tool kit combines several tools for compiler and WCET analysis integration, tool configuration and flow fact transformation. We demonstrated the platin tool kit on a sample application and showed how to perform a WCET analysis using platin. Due to its design, it should be possible to adapt and integrate platin into other LLVM based compilers with a low effort.

Acknowledgement

This work was partially funded under the European Union's 7th Framework Programme under grant agreement no. 288008: Time-predictable Multi-Core Architecture for Embedded Systems (T-CREST). This paper is based on [11]. We would like to thank Christoph Cullmann and Gernot Gebhard for their work on the Patmos port of the AbsInt aiT analyser.

References

1. Abbaspour, S., Brandner, F., Schoeberl, M.: A time-predictable stack cache. In: Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2013 IEEE 16th International Symposium on. pp. 1–8 (June 2013)
2. Degasperi, P., Hepp, S., Puffitsch, W., Schoeberl, M.: A method cache for Patmos. In: Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2014 IEEE 17th International Symposium on. pp. 100–108 (June 2014)
3. Falk, H., Lokuciejewski, P.: A compiler framework for the reduction of worst-case execution times. Real-Time Systems pp. 1–50 (2010)
4. Gomony, M.D., Garside, J., Akesson, B., Audsley, N., Goossens, K.: A generic, scalable and globally arbitrated memory tree for shared DRAM access in real-time systems. In: Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition. pp. 193–198. DATE '15, EDA Consortium, San Jose, CA, USA (2015), `http://dl.acm.org/citation.cfm?id=2755753.2755795`
5. Goossens, S., Kuijsten, J., Akesson, B., Goossens, K.: A reconfigurable real-time SDRAM controller for mixed time-criticality systems. In: Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis. pp. 2:1–2:10. CODES+ISSS '13, IEEE Press, Piscataway, NJ, USA (2013), `http://dl.acm.org/citation.cfm?id=2555692.2555694`
6. Huber, B., Prokesch, D., Puschner, P.: Combined WCET analysis of bitcode and machine code using control-flow relation graphs. In: Proceedings of the 14th ACM SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems. pp. 163–172. The Association for Computing Machinery (2013)
7. Kasapaki, E., Spars, J.: Argo: A time-elastic time-division-multiplexed noc using asynchronous routers. In: Asynchronous Circuits and Systems (ASYNC), 2014 20th IEEE International Symposium on. pp. 45–52 (May 2014)
8. Kirner, R., Puschner, P., Prantl, A.: Transforming flow information during code optimization for timing analysis. Real-Time Systems 45, 72–105 (2010)

9. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization (CGO'04). pp. 75–88. IEEE Computer Society (2004)
10. Lisper, B.: Sweet a tool for wcet flow analysis (extended abstract). In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications, Lecture Notes in Computer Science, vol. 8803, pp. 482–485. Springer Berlin Heidelberg (2014), http://dx.doi.org/10.1007/978-3-662-45231-8_38
11. Puschner, P., Prokesch, D., Huber, B., Knoop, J., Hepp, S., Gebhard, G.: The T-CREST approach of compiler and WCET-analysis integration. In: Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2013 IEEE 16th International Symposium on. pp. 1–8 (June 2013)
12. Schoeberl, M., Brandner, F., Hepp, S., Puffitsch, W., Prokesch, D.: Patmos reference handbook. Tech. rep. (2015), http://patmos.compute.dtu.dk/patmos_handbook.pdf
13. Schoeberl, M., Schleuniger, P., Puffitsch, W., Brandner, F., Probst, C.W., Karlsson, S., Thorn, T.: Towards a time-predictable dual-issue microprocessor: The Patmos approach. In: First Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011). pp. 11–20 (March 2011)
14. Schoeberl, M., Silva, C., Rocha, A.: T-Crest: A Time-Predictable Multi-Core Platform For Aerospace Applications. ESA - SP, European Space Agency, ESA (2014)

A Certifiable Domain-Specific Language for Reasoning about Trust Aggregation

Michael Huth and Jim Huan-Pu Kuo

Department of Computing, Imperial College London
London, SW7 2AZ, United Kingdom
{m.huth, jimhkuo}@imperial.ac.uk

Abstract. It is increasingly important to analyze system security quantitatively using concepts such as trust, reputation, cost, and risk. This requires a thorough understanding of how such concepts should interact so that we can validate the assessment of threats, the choice of adopted risk management, and so forth. To this end, we propose a declarative language *Peal+* in which the interaction of such concepts can be rigorously described and analyzed. *Peal+* has been implemented in tool *PEALT* using the SMT solver Z3 as the analysis back-end. *PEALT*'s code generators target complex back-ends and evolve with optimizations or new back-ends. Thus we can neither trust the tool chain nor feasibly prove correctness of all involved artefacts. We eliminate this need for trust by independently certifying scenarios found by back-ends in a manner agnostic of code generation and choice of back-end. This scenario validation is compositional, courtesy of Kleene's 3-valued logic and potential refinement of scenarios. We prove the correctness of this validation, discuss how *PEALT* presents scenarios to further users' understanding, and demonstrate the utility of this approach by showing how it can express attack-countermeasure trees so that the interaction of attack success probability, attack cost, and attack impact can be analyzed.

1 Introduction

It is well recognized that the analysis of threats to system security goes beyond the exposure and fixing of vulnerabilities and that it has to take account of contextual influences such as risks, trust assumptions, the reputation of domains, and so forth. However, it is often not clear how such different concepts interact in the threat space (which the attacker controls) or how they should interact in a system design space (which the designer thinks he controls). For example, when the Heartbleed vulnerability became known even security experts could not agree on whether users should immediately change their passwords on web accounts that used versions of OpenSSL vulnerable to this attack [20]. In particular, it was difficult to know whether the account was compromised, and renewing a password in a compromised account might leak that password to an attacker.

In general, threat analysts have a host of techniques and models at their disposal that allow them to assess security threats, let us mention here attack trees

[19, 13, 12] and Stackelberg games for security (see e.g. [10]) as two prominent examples. Also, probabilistic risk analysis [2] offers a rich set of tools that threat analysts may use to study the interaction of factors that influence security. Alas, tools from risk analysis view attackers as passive environments (e.g. modeling mean time between failures of a hard disk) and not as active agents (e.g. a cyber terrorist who seeks access to programmable logic controllers in a SCADA system). We therefore would like support for modeling the interaction of concepts such as protection cost, impact of successful attacks, perception of risk, reputation of agents, and so forth, in a system exposed to active attacks. The active nature of attackers suggests to model action and reaction with AND/OR structures, e.g. as present in two-person games or first-order logic. The desire to study interaction of quantitative concepts suggests use of an expressive logical language with appropriate theories for reals; expressiveness means we can easily extend studies to new concepts or interaction modes, and theories enable us to do correct quantitative reasoning. We cannot assume, though, that threat analysts are trained logicians, so we require automated reasoning support for such logics to build *auto-interactive verifiers*. SMT solvers, e.g. Z3 [14], thus look like apt vehicles for expressing and analyzing such interaction in this manner.

Choosing an SMT solver as the back-end also poses problems. Its input language is too complex and universal, but security analysts prefer languages specific to their modeling domain. For code generators from domain-specific languages into SMT back-ends we need assurance that results computed by back-ends are correct and sensible in modeled domains. Security analysts want results communicated in forms appreciable to them. Finally, users may formulate conditions that are vacuously true, or vacuously false in the modeled domain. This may identify a specification error or may instead validate that an analyst has realized an important invariant – e.g. that the risk is always below an acceptable threshold. Our paper presents results that directly address these problems.

Figure 1 shows how our contributions reported in this paper are realized in the tool $PEALT$. Users specify $Peal+$ conditions to be analyzed, and domain-specific knowledge or assumptions; $PEALT$ converts these specifications into Z3 code which the SMT solver Z3 solves; the raw output of Z3 results is then post-processed and analyzed over the $Peal+$ conditions; and feedback is reported so that all scenarios are certified. The user may then inspect that feedback and either be satisfied or edit conditions or domain specifics for further analysis.

Outline of paper. In Section 2, we present language $Peal+$ in which threats can be modeled and analyzed when quantitative information contains non-deterministic uncertainty; and we discuss automated vacuity checking. In Section 3, we discuss how the implementation of $Peal+$, its analyses, and certification are supported by the use of partial evaluation to render certified scenarios to users in compact ways that should facilitate users' comprehension. In Section 4, we present our algorithm that independently certifies that scenarios computed from analyses by back-ends such as Z3 are correct for the modeled problem – eliminating the need to trust our code generation methods or back-ends. In Section 5, we show $Peal+$'s utility as an intermediate language for analyzing

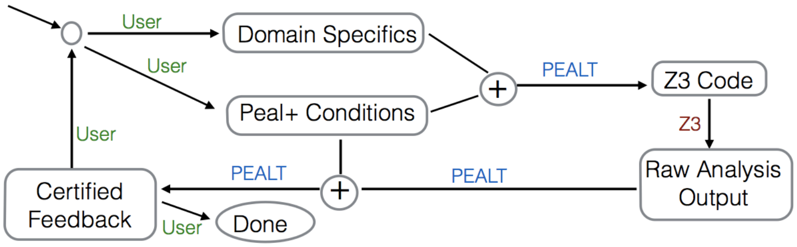


Fig. 1. Overview of our approach of auto-interactive verification with tool *PEALT*, showing activities done by users, by *PEALT* or by its back-end SMT solver Z3

the interaction of threat concepts in tree-like models. In Section 6, we further discuss and evaluate language *Peal+* and its implementation in tool *PEALT*. Section 7 features related work, and Section 8 concludes the paper.

2 Domain-specific language and its vacuity checks

Figure 2 shows a formal grammar for our language *Peal+* that can express interaction of security aspects as well as the logical/quantitative analysis of such interaction. *Peal+* shares the coarse structure of its predecessor *Peal* [4, 7]: rules condition a score on a predicate, policies are built from rules, policy sets are built out of policies, conditions are formed out of policy set comparisons; and analyses have conditions as arguments. The meaning of analysis types is the intuitive one of their names seen in Figure 2. The meaning of conditions is given by that of propositional logic and of comparison operators over reals. Thus it suffices to define how policy sets evaluate to reals in an environment in which all predicates have truth values, all real variables have a real value, and all non-deterministic uncertainties are resolved – so all scores evaluate to a real number.

We state this semantics informally here, and formally in Figure 3. A rule *rule* returns its declared score when its declared predicate is true; otherwise, it has no effect. The meaning of a policy is then given as follows: if none of its rules has a true predicate, its meaning is that of its default score; otherwise, its meaning is obtained by first computing the meaning of all scores from its rules with true predicates, and then applying operator *op* to that set of computed reals.

The grammar for scores allows us to write expressions such as 0.45, -124.5, $0.67 * x$, $0.5 * p.\mathbf{sc}$, $0.4 * x\ [-0.1, 0.1]$, or $0.5 * p.\mathbf{sc}\ [-0.05, 0.05]$ where x is a real-valued identifier and p is a policy set. In a given environment, the meaning of scores without intervals $[l, u]$ is that of normal arithmetic with variable values given by the environment. The meaning of expressions $s\ [l, u]$ is $x + y$ where x is the meaning of s in the environment, and y from $[l, u]$ is the non-deterministic choice of the environment from interval $[l, u]$. The latter may, e.g., express the level of confidence that an expert has in choosing a subjective probability s. To ensure consistency, we require $l \leq 0.0 \leq u$. For example, $u < l$ would be logically inconsistent and $l > 0$ would suggest to change $s\ [l, u]$ to the equivalent but

$$
\begin{aligned}
\mathit{alys} &::= \textbf{satisfiable}?\ \mathit{cond}\ \mid \textbf{always\_true}?\ \mathit{cond} \mid \textbf{always\_false}?\ \mathit{cond}\\
&\ \textbf{equivalent}?\ \mathit{cond}\ \mathit{cond}\ \mid \textbf{different}?\ \mathit{cond}\ \mathit{cond} \mid \textbf{implies}?\ \mathit{cond}\ \mathit{cond}\\
\mathit{cond} &::= q \mid \neg \mathit{cond} \mid \mathit{cond} \,||\, \mathit{cond} \mid \mathit{cond} \,\&\&\, \mathit{cond} \mid pSet \leq pSet \mid pSet < pSet
\end{aligned}
$$

$$
\begin{aligned}
\mathit{op} &::= \mathit{min} \mid \mathit{max} \mid + \mid *\\
pSet &::= \mathit{pol} \mid \mathit{op}\ (pSet,\ pSet)\\
\mathit{pol} &::= \mathit{op}\,(\mathit{rule}^*)\ \textbf{default}\ \mathit{score}\\
\mathit{rule} &::= (q\ \mathit{score})
\end{aligned}
$$

$$
\begin{aligned}
\mathit{score} &::= rawScore \mid rawScore\ [realConst\,,\ realConst]\\
rawScore &::= realConst \mid realVar \mid realConst * realVar\\
realVar &::= \mathit{identifier} \mid pSet.\textbf{sc}
\end{aligned}
$$

Fig. 2. Syntax of *Peal+* where q ranges over some language of predicates; constants and variables occurring in *score* expressions range over real numbers, and $[realConst\,,\ realConst]$ ranges over closed real intervals. For sake of clarity, keywords of *Peal+* are written in boldface here, e.g., $pSet.\textbf{sc}$ denotes the score of $pSet$

$$
\begin{aligned}
\mathcal{E}_e(\mathit{op}(pS_1, pS_2)) &= \mathit{op}(\mathcal{E}_e(pS_1), \mathcal{E}_e(pS_2))\\
\mathcal{E}_e(\mathit{op}((q_1\ s_1)\ \ldots (q_n\ s_n))\ \mathit{default}\ s) &= \mathit{op}(Z) && (\text{if } Z \neq \emptyset)\\
\mathcal{E}_e(\mathit{op}((q_1\ s_1)\ \ldots (q_n\ s_n))\ \mathit{default}\ s) &= \mathcal{E}_e(s) && (\text{if } Z = \emptyset)
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}_e(a) &= a && (\text{constant } a)\\
\mathcal{E}_e(x) &= e(x) && (x \text{ not of form } p.\textbf{sc})\\
\mathcal{E}_e(a * x) &= a \cdot e(x) && (x \text{ not of form } p.\textbf{sc})\\
\mathcal{E}_e(p.\textbf{sc}) &= \mathcal{E}_e(p) && (\text{evaluate policy } p)\\
\mathcal{E}_e(a * p.\textbf{sc}) &= a \cdot \mathcal{E}_e(p) && (\text{evaluate policy } p)\\
\mathcal{E}_e(r\ [l,u]) &= \mathcal{E}_e(r) + e(p, q_i, [l,u]) && ([l,u] \text{ declared in } p \text{ for predicate } q_i)\\
\mathcal{E}_e(r\ [l,u]) &= \mathcal{E}_e(r) + e(p, \mathit{default}, [l,u]) && ([l,u] \text{ declared in default score } s \text{ of } p)
\end{aligned}
$$

Fig. 3. Semantics $\mathcal{E}_e(pSet)$ of policy sets (acyclic as in Def. 1), given an environment e that maps predicates to truth values, scores to reals, and resolves non-deterministic choice of uncertainty intervals. Scores r range over raw scores, a over constants, x over variables, and $p.\textbf{sc}$ over policy scores. Set Z equals $\{\mathcal{E}_e(s_i) \mid 1 \leq i \leq n, e(q_i) = \mathit{true}\}$

more comprehensible $(s + l)$ $[0.0, u - l]$ when $l \leq u$. The meaning of variable *pSet*.**sc** is that of *pSet* computed by the operational semantics just described. For this to be well-defined, the set of declared policy sets must not create cyclic dependencies in *Peal*+:

Definition 1. *Let* p_1 *and* p_2 *be in a set* $\mathcal{P}$ *of Peal+ policy sets. Then* p_1 depends on p_2 *(written* $p_2 \prec p_1$*) if there is a score s in* p_1 *that contains or equals variable* p_2.**sc**. *Set* $\mathcal{P}$ *is* acyclic *if the transitive closure of* $\prec$ *over* $\mathcal{P} \times \mathcal{P}$ *is acyclic.*

Peal+ extends *Peal* in important ways: scores may have variables and non-deterministic uncertainty, policy sets have the same composition operators as policies, conditions subsume propositional logic and may compare policy sets, and the result of a policy set can be referred to as variable within a score expression. With these extensions, *Peal*+ is expressive enough to capture metrics, tree-like models, cost functions, and basic probabilistic computations.

Let us illustrate the use of *Peal*+ with an example modeling risks that a car rental company may face when renting out cars to clients. Figure 4 shows how rules, policies, policy sets, and conditions for this example are declared in the input language of our tool *PEALT*. Declarations are divided into blocks by keywords such as `POLICIES` and lines that begin with % are used for comments.

A notable feature of the tool input language is the declaration block `DOMAIN_SPECIFICS` in which specifiers can enter code from the input language of the SMT solver Z3 [14] to further constrain the model. This would typically be used to express assumptions or knowledge of the modeled domain, and uses Z3 syntax since Z3 is the current back-end of our tool. For example, the second assertion in Figure 4 uses this to express that luxury cars must not be rented out for off-road driving. It represents risk and trust as values in $[0, 1]$, and uses $f(x) = 1 - x$ to convert one into the other. More sophisticated relationships between trust and risk may be captured in *Peal*+ as well. This *Peal*+ model is conceptually similar to the use of score cards that assess risks in mortgage applications [17]. Next, we discuss vacuity checking and how we support this.

Vacuity checking. The analyses `always_true`? and `always_false`? reduce to satisfiability checks but their intent is to check for so called vacuities [11]: a condition that is always true or always false may be a specification error (as in temporal logic verification of hardware [11]), evidence for a desired invariant, or may require further scrutiny of the specifier. Our tool automatically enforces both types of vacuity check on *all* declared conditions. The reason is that declared conditions are likely to contribute to input of a declared analysis, and so we want to alert users to those conditions that are vacuously true, resp. false.

For example, condition `c1` of the Car Rental Risks example in Figure 4 is reported to be always true, so the "insurance risk" which multiplies monetary loss with its associated risk is never above 50,000. If Z3 can't decide a vacuity check (output `UNKNOWN`), *PEALT* reports checked conditions as "may be" vacuities. *PEALT* only reports names of vacuously true or false conditions. Users who want more detailed feedback as described below need to "promote" such a vacuity

```
POLICIES
% policy capturing risk of financial loss dependent on type of rented car
b1 = max ((isLuxuryCar 150000) (isSedan 60000) (isCompact 30000)) default 50000
% policy capturing trust in rentee dependent on type of his or her driving license
b2 = min ((hasUSLicense 0.9) (hasUKLicense 0.6) (hasEULicense 0.7)
          (hasOtherLicense 0.4 [-0.1,0.1])) default 0
% policy that captures potential risk dependent on type of intended car usage
% this policy happens not to be used in the conditions below
b3 = max ((someOffRoadDriving 0.8) (onlyCityUsage 0.4) (onlyLongDistanceUsage 0.2)
          (mixedUsage 0.25)) default 0.3
% policy that accumulates some signals that may serve as additional trust indicators
b4 = + ((accidentFreeForYears 0.05*x) (speaksEnglish 0.05) (travelsAlone -0.2)
        (femaleDriver 0.1)) default 0
% convert trust b2 into risk b2 using f(x) = 1-x
b2_risk = +((True 1.0) (True -1*b2_score)) default 0.0
POLICY_SETS
% casting b2_risk into policy set
pSet0 = b2_risk
% policy set that multiplies risk with potential financial loss
pSet1 = *(b1,pSet0)
% casting policy p4 into a policy set
pSet_b4 = b4
CONDITIONS
% condition that the risk aware potential financial loss is below a certain bound
c1 = pSet1 <= 50000
% condition that the accumulated trust is above a certain threshold
c2 = 0.4 < pSet_b4
% condition that insists that two previous conditions have to hold
c3 = c1 && c2
DOMAIN_SPECIFICS
% real x models accident-free years of driving, 'truncated' at value 10
(assert (and (<= 0 x) (<= x 10)))
% capturing a company policy: luxury cars must not be used for off road driving
(assert (implies (isLuxuryCar (not someOffRoadDriving))))
% capturing that the different types of rental cars are mutually exclusive
(assert (and (implies isLuxuryCar (and (not isSedan) (not isCompact)))
             (implies isSedan (and (not isLuxuryCar) (not isCompact)))
             (implies isCompact (and (not isSedan) (not isLuxuryCar)))))
% capturing that cars that are only used in cities are not used in a mixed sense
(assert (implies onlyCityUsage (not mixedUsage)))
% capturing that cars used only for longdistance driving are not used in a mixed sense
(assert (implies onlyLongDistanceUsage (not mixedUsage)))
% capturing domain constraints (or company policy?) that city driving cannot happen off road
(assert (implies onlyCityUsage (not someOffRoadDriving)))
% capturing that cars used only for longdistance driving must drive off road
(assert (implies onlyLongDistanceUsage (not someOffRoadDriving)))
ANALYSES
% is condition c1 always true? this would suggest an invariant
name1 = always_true? c1
% is condition c3 always true? this would suggest a specification error
name2 = always_true? c3
```

Fig. 4. *Peal+* model of Car Rental Risks

analysis into the ANALYSES section, where more detailed feedback is provided. Users may turn automated vacuity checking on or off under "Settings". We recommend vacuity checks to be done at least once for model validation.

3 Feedback for users

As described in [7], we extract raw Z3 output and render it in pretty printed form, as seen in the initial part of Figure 5. But for larger case studies, it becomes hard to digest even pretty printed information: one often cannot see the forest for all the trees. So we now also output for each analysis a summary of the scenario, its certification (detailed in Section 4), and supporting information. Figure 5 shows typical such output for the Car Rental Risks example. Scenarios also report any non-deterministic choices of uncertainty as seen for variable `b2_hasOtherLicense_U` – which functions as t_2 in $eval(0.4\,[-0.1, 0.1], env)$ as detailed in Figure 7 – in that figure. PEALT reports the certification outcome and

```
Result of analysis [name2 = always_true? c3]
c3 is (pSet1 <= 50000.0) && (pSet_b4 > 0.4)

c3 is NOT always true, for example, in the scenario in which:
accidentFreeForYears is True, femaleDriver is True, isLuxuryCar is True,
mixedUsage is True, speaksEnglish is True, travelsAlone is True, ...
hasEULicense is False, hasOtherLicense is False, hasUKLicense is False, ...,
b1_score is 150000, b2_hasOtherLicense_U is 0, b2_risk_score is 1, ...

Certification of analysis [name2] succeeded.
Additional predicates set to false for certification: Set(hasUSLicense, hasEULicense)

Policy scores statically inferred in this certification process:
b1 has score 150000, b2 has score 0.6, b2_risk has score 0.4,
b3 has score 0.25, b4 has score 0.55

Policies in analysis [name2] partially evaluated in certified scenario:
b1 = max (([isLuxuryCar] 150000)) default 50000 ...
b4 = + (([accidentFreeForYears speaksEnglish] 0.55)) default 0
```

Fig. 5. Output format of analyses (hand-edited to save space): scenario (if applicable), certification status and possible refinements, policy scores inferred during certification, and policies partially evaluated in certified scenario.

refinements of predicates and real variables that certification may have brought about (when applicable), lists scores of *all* policies that certification could statically infer, and then partially evaluates *only relevant* policies (not for `b3` in Figure 5) over the successfully certified scenario to then display them in this more compact and meaningful manner. For the latter, true predicates are grouped within square brackets and reported with aggregated score in red (colors not shown in figure), as this is the score for the policy as well. Rules with false predicates aren't shown; in particular, if all predicates are false, an empty policy with red default score is shown. Rules whose predicates have unspecified truth values are shown individually (in green) where "?" marks them as *don't care* rules.

PEALT uses Z3's `push` and `pop` constructs for incremental solving of more than one analysis. The efficiency may also raise usability issues: the output in Figure 5 was obtained after all other analyses were commented out. If we run all these analyses in their declared sequence, however, the scenario reported for `name2` will be different. Similar effects may happen when automated vacuity checking changes its OFF/ON status. On the other hand, this seems at worst to make the user temporarily confused and so we don't think this issue is serious enough to give up the efficiency gains of using the `push` and `pop` constructs.

4 Scenario certification

Users from high-assurance domains need compelling evidence that scenarios computed by back-ends from code *PEALT* generates are valid for analyzed *Peal+* conditions, and they want to be able to relate scenarios to conditions in a comprehensible manner. We report additional support for the latter below. As for the former, what if our Z3 code generation method contains logical mistakes? What if we make wrong assumptions about the operation of the tool Z3? What if some Z3 features we use contain implementation flaws? We think these questions make a compelling case for independently proving the validity of a scenario discovered for a *Peal+* condition; we refer to such independent proof as *certification.* Back-ends such as Z3 compute scenarios that are very compact in that they don't define values for some variables. Certification is therefore non-trivial as it has to reason that these are indeed "don't care" variables. Such a certification should be comprehensible to non-experts and efficient – giving it the flavor of an NP problem although the underlying decision problems may be undecidable. We propose a compositional certification of don't care variables that may lose precision and so may have an inconclusive output. In the latter case, one of the predicates of the scenario may not have a specified truth value. We then set that value to *false* and repeat the certification algorithm on this refined scenario. This process is efficient as it examines conditions compositionally and greedily refines scenarios until it succeeds or not. Refined predicates are set to *false* and *not* to *true*: users want to see as few trees in the forest as possible, and false predicates only have an effect in a policy when all its predicates are false.

This certification process represents a scenario, a model returned by Z3, as a function I that maps real variables to real numbers or $\perp$, and predicates to *true*, *false* or $\perp$. Symbol $\perp$ models that the scenario did not specify a value for the variable in question. For predicates, $\perp$ ("unknown") is also the third truth value of Kleene's 3-valued logic [9]. Figure 5 shows how *PEALT* reports a scenario for analysis `name2` from Figure 4. To explain our certification, we need to define the refinement of environments, which are all well typed in that they map any variable either to value $\bot$ or a value of its declared data type – Real or Boolean.

Definition 2. *Let env_1, env_2 be environments over a set of variables $\mathcal{V}$. Then env_2* refines *env_1 if for all x in $\mathcal{V}$, $env_1(x) \neq \bot$ implies $env_1(x) = env_2(x)$.*

This means that refinements can change $\bot$ values of variables to any value of their declared data type Real or Boolean, but they cannot change non $\bot$ values.

The function *recursivelyCertify*, depicted in Figure 6, is first called as $recursivelyCertify(c, I, v, \emptyset)$ which checks whether condition c has claimed truth value v in the scenario/Z3 model I. It outputs *true* if this claim could be certified, *false* if a *logical* flaw in the claim was detected, and outputs $\bot$ otherwise. Wrapper function $certifyWrapper(c, I, v)$ in Figure 6 converts *true*, *false* and $\bot$ into certification *success*, *failure*, and *inconclusive*, respectively.

```
$certifyWrapper(c, I, v)$ {  % condition $c$, scenario $I$, and $v$ in $\{false, true\}$
    if ($recursivelyCertify(c, I, v, \emptyset)$ == true) { return success; }
    if ($recursivelyCertify(c, I, v, \emptyset)$ == false) { return failure; }
    elseif { return inconclusive; }
}

$recursivelyCertify(c, env, v, cp)$ {  % returns true, false or $\bot$
    $cp' = collectCertifiablePolicyScores(env)$;
    $env' = env + cp'$;  % program point L1: extend $env$ with bindings of $cp'$
    $o = certCond(c, env', v)$;
    if ($o == \bot$) {
        if ($cp \neq cp'$) {
            return $recursivelyCertify(c, env', v, cp')$;
        } elsif ($\exists q \colon env'(q) = \bot$) {
            pick one $q$ with $env'(q) = \bot$;
            $env' = env' + [q \mapsto false]$;
            return $recursivelyCertify(c, env', v, cp')$;
        } else {return $o$;} % triggers exception upstream (not shown here)
    } else {  % program point L2
        return $o$;  % output true means success, false means failure
    }
}
```

Fig. 6. Function $recursivelyCertify(c, I, v, \emptyset)$ checks whether condition c has claimed truth value v in empty hash map *cp* (written $\emptyset$) and scenario I where it may refine the latter. Function *certifyWrapper* wraps this into success, failure, or inconclusive result

The truth value v used in $recursivelyCertify(c, I, v, \emptyset)$ is determined by the type of analysis. For example, if `always_false? c` returns `SAT`, it means the found scenario should be evidence for c being true, and so v equals *true*. The treatment of analyses with two arguments is similar. For example, for a `SAT` outcome of `implies? c1 c2`, the scenario should be evidence for c_1 being true and c_2 being false. So we need to achieve two certifications, $\mathit{recursivelyCertify}(c_1, I, \mathit{true}, \emptyset)$ and $\mathit{recursivelyCertify}(c_2, I, \mathit{false}, \emptyset)$ for this.

Function *recursivelyCertify* refines I into an environment env' by setting predicates to *false* or adding a statically inferred score to a policy. The latter means that environments are not only defined on predicates and real variables but may also map policy names to their inferred scores. At program point l2, such static inference of policy scores is delegated to function *collectCertifiablePolicyScores* in Figure 7. In this extended environment env', function *certCond*, shown in Figure 8, determines the truth value of the condition in that environment under Kleene's 3-valued logic [9]. If that value is $\perp$, we call *recursivelyCertify* again but with a refined environment that either inferred at least one new policy score or set a predicate to false. If the truth value of the condition is $\neq \perp$, function *recursivelyCertify* outputs that value.

Parameter *cp* is a hash map that has policies as keys and their statically inferred scores as values. We check "progress" of *cp* since static inference of a policy score may then enable more such inferences for other policies. Function $\mathit{collectCertifiablePolicyScores}(\mathit{env})$ initializes in *cp* an empty hash map. For each declared policy *pol* it stores in *score* the output of function $\mathit{certPolicy}(\mathit{pol}, \mathit{env})$ depicted in Figure 7. Thus we statically infer the score of *pol* (rather than consulting $\mathit{env}(\texttt{p\_score})$ if that were $\neq \perp$), so that policy scores are certified before their use in certification of policy scores they depend on. Then either an equality check of $\mathit{certPolicy}(\mathit{pol}, \mathit{env})$ and $\mathit{env}(\mathit{p\_score})$ is performed – whose failure will fail certification – or we check whether the static analysis returns a real value (i.e. not $\perp$), in which case we extend the hash map so that key *pol* has value *score*. Finally, the hash map is returned.

Function $\mathit{certPolicy}(\mathit{pol}, \mathit{env})$ first checks whether some predicate q within policy *pol* has unspecified truth value in environment *env*. If so, it returns $\perp$ since the score of *pol* cannot be determined. Otherwise, if all predicates in *pol* are false in environment *env*, the default case applies and the evaluation of the default score s in environment *env* is returned. Finally, if some predicates in *pol* are true (and none are then false), we return the application of *op* to the evaluation $\mathit{eval}(s_i, \mathit{env})$ of all "true" score expressions s_i in environment *env*.

Function $\mathit{eval}(s, \mathit{env})$ has two types of input for s depending on whether s is a raw score t_1 or contains an uncertainty interval $[l, u]$ that we translate into Z3 code as a real variable t_2. This function does a static analysis that consults $\mathit{env}(p)$ when evaluating variables of form $p.\mathbf{sc}$ and consults $\mathit{env}(x)$ for all other variables x. This consults $\mathit{env}(p)$ and not $\mathit{env}(p.\mathbf{sc})$ so that policy scores get certified based on certified scores of policies that they depend upon. Although $\perp$ is strict for $+$, we relax its strictness for $*$ in expressions $a * x$ when a evaluates to 0.0, in which case $a * x$ also evaluates to 0.0.

Last, but not least, we turn to function $\mathit{certCond}(c, \mathit{env}, v)$ in Figure 8. It compositionally evaluates over the structure of c whether this condition computes to truth value v in environment *env*. This makes use of 3-valued propositional logic of Kleene [9], where for example $\perp \vee x = x$ and $\neg\perp = \perp$. The intuition is that $\perp$ stands for either *true* or *false* and that equations are valid

```
$collectCertifiablePolicyScores(env)$ {
% returns hash map of some policies, with their statically inferred scores as keys
    $cp = \emptyset$;
    $for$ ($all\ declared\ policies\ pol$) {
        $score = certPolicy(pol, env)$;
        $if$ ($env(pol\_score) \neq \bot$){
            $if$ ($score \neq env(pol\_score)$){
                $report\ certification\ exception$; $break$;
            }
        }
        $if$ ($score \neq \bot$) {$cp = cp + [pol \mapsto score]$; }
    }
    $return\ cp$;
}

$certPolicy(pol, env)$ {  % returns statically inferred policy score or $\bot$
    $if$ ($\exists (q_i\, s_i) \in pol\colon env(q_i) = \bot$) { $return\ \bot$; }
    $elseif$ ($X^{pol}_{env} == \emptyset$) { $return\ \ eval(s, env)$; }
    $else$ { $return\ op(X^{pol}_{env})$; }
}

$eval(s, env)$ {
% $s = t_1$ or $s = t_1 + t_2$ with $t_1$ being constant $a$, variable $x$ or product $a * x$
% and $t_2$ being variable $x$ not of form $p.\mathbf{sc}$ (modeling uncertainty)
    $if$ ($t_1$ $of\ form\ a$) {$acc = a$; }
    $elseif$ ($t_1$ $of\ form\ p.\mathbf{sc}$) {$if$ ($env(p) \neq \bot$) {$acc = env(p)$; } $else$ {$return\ \bot$; }}
    $elseif$ ($t_1$ $of\ form\ x$) {$if$ ($env(x) \neq \bot$) {$acc = env(x)$; } $else$ {$return\ \bot$; }}
    $elseif$ ($t_1$ $of\ form\ a * p.\mathbf{sc}$) {
        $if$ ($a == 0.0$) {$acc = 0.0$; }
        $elseif$ ($env(p) \neq \bot$) {$acc = a * env(p)$; }
        $else$ {$return\ \bot$; }        % here $a$ non-zero but $env(p)$ equals $\bot$
    }
    $elseif$ ($t_1$ $of\ form\ a * x$) {        % here $x$ is not of form $p.\mathbf{sc}$
        $if$ ($a == 0.0$) {$acc = 0.0$; }
        $elseif$ ($env(x) \neq \bot$) {$acc = a * env(x)$; }
        $else$ {$return\ \bot$; }        % here $a$ non-zero but $env(x)$ equals $\bot$
    }
    $if$ ($s$ $of\ form\ t_1 + t_2$) {
        $if$ ($env(t_2) \neq \bot$) {$acc = acc + env(t_2)$; }
        $else$ {$return\ \bot$; }        % here $env(t_2)$ equals $\bot$, strict for $+$
    }
    $return\ acc$;
}
```

Fig. 7. Function *collectCertifiablePolicyScores*(*env*) returns hash map for policies *pol* with values *score* statically inferred as result of *pol* in *env*. Function *certPolicy* certifies whether the score of policy *pol* of the form *op* $((q_1\ s_1) \ldots (q_n\ s_n))$ *default* s or *op* () *default* s in environment *env* is inferable. Set X^{pol}_{env} denotes $\{eval(s_i, env) \mid env(q_i) = true\}$ and function $eval(s, env)$ statically infers the value of score s in environment *env*

```
$certCond(c, env, v)$ {  % returns $true$, $false$ or $\bot$; comparisons to $\bot$ return $\bot$
    $if\ (c\ of\ form\ q)$ { $return\ (v == env(q));$ }
    $elseif\ (c\ of\ form\ \neg c_1)$ { $return\ certCond(c_1, env, \neg v);$ }
    $elseif\ (c\ of\ form\ (c_1 \wedge c_2))$ { $if\ (v == true)$ {$lop = \wedge;$ } $else$ {$lop = \vee;$ }
        $return\ certCond(c_1, env, v)\ lop\ certCond(c_2, env, v);$ }
    } $elseif\ (c\ of\ form\ (c_1 \vee c_2))$ { { $if\ (v == false)$ {$lop = \wedge;$ } $else$ {$lop = \vee;$ }
        $return\ certCond(c_1, env, v)\ lop\ certCond(c_2, env, v);$ }
    } $elseif\ (c\ of\ form\ (pS_1 \leq pS_2))$ {
        $if(v == true)$ { $return\ certPSet(pS_1, env) \leq certPSet(pS_2, env);$ }
        $else$ { $return\ certPSet(pS_2, env) < certPSet(pS_1, env);$ }
    } $elseif\ (c\ of\ form\ (pS_1 < pS_2))$ {
        $if(v == true)$ { $return\ certPSet(pS_1, env) < certPSet(pS_2, env);$ }
        $else$ { $return\ certPSet(pS_2, env) \leq certPSet(pS_1, env);$ }
    }
}

$certPSet(pSet, env)$ {  % returns $true$, $false$ or $\bot$; if $env(pol)$ not found, returns $\bot$
    $if\ (pSet\ of\ form\ pol)$ {$return\ env(pol);$ }
    } $elseif\ (pSet\ of\ form\ op(pS1, pS2))$ { $return\ op(certPSet(pS_1, env), certPSet(pS_2, env));$ }
}
```

Fig. 8. Function $certCond(c, env, v)$ decides whether condition c has truth value v in environment env, and $certPSet(pSet, env)$ covers this for policies and their composition

under this interpretation. This is an abstraction as $q \vee \neg q$ evaluates to $\bot$ in this logic whenever q has value $\bot$. We note that $\bot$ is strict for comparison operators $==$, $\leq$, and $<$ in function *certCond*. If the condition c is atomic q, we check whether claimed truth value v matches what the environment says about q. If c is $\neg c_1$, we reduce this to the certification that c_1 has the negated truth value $\neg v$ in the same environment. The cases of conjunction and disjunction are dual and need to consider whether v equals *true* or *false*. This structure is also seen in comparing policy sets in a condition, which compares their scores as computed by the environment in function *certPSet* ($\bot$ indicates no score is present).

The correctness theorem for certification refers to the meaning of *Peal+* in environments where all variables have a value from their declared data type Real or Boolean. This operational semantics was given in Section 2 and Figure 3.

Theorem 1. *Let c be a Peal+ condition such that the set of policy sets occurring in c is acyclic. Let v be a truth value true or false. Let I be a scenario produced for c from a back-end. Let function $recursivelyCertify(c, I, v, \emptyset)$ return true and let env' be the value of this environment at program point L2. Let env'' refine env' such that env'' maps no variable to $\bot$. Then condition c evaluates to v in environment env'' under the operational semantics of Peal+.*

Proof (Sketch). We only have to show the claim for function *certCond*, given the code structure of *recursivelyCertify*. The claim is proved using structural induction over the condition c, noting that sub-conditions also have acyclic sets of policy sets. The cases rely on that fact that $\perp$ is strict for all algebraic operators with the noted exception of $eval(0.0 * \perp, env) = 0.0$.

The cases that compare two policy sets require proof of an auxiliary lemma: *"Whenever the output of* $certPSet(pS, env')$ *is not equal to* $\perp$*, then that output is the score of policy set pS in all environments that refine environment* env'*."* This is shown for policies and composed policy sets by structural induction.

For the first case of a policy set being a policy, we require a second auxiliary lemma: *"Let pol be a policy and* env' *an environment such that* $env'(pol)$ *is not equal to* $\perp$*. Then* $env'(pol)$ *is the score of policy pol in all environments that refine environment* env'*."* The proof of this lemma appeals to the linear order in which statically inferred scores of policies are added as hash values, where env' is of form $env + cp'$ as seen at program point L1 in function *recursivelyCertify*. Since the set of policies occurring in condition c is acyclic, this order is indeed well founded and so we can use well founded induction to prove this lemma. □

The above theorem says that successful certification of the computed environment env' means that all "completions" of env' that resolve $\perp$ values with any legal value of the respective data type will compute the claimed truth value for the condition in question. In particular, variables x with $env'(x) = \perp$ are genuine "don't care" variables for this successful certification.

Our certification runs in polynomial time in its input: the number of recursions is bounded by $m + n$, with m the number of declared policies and n the number of predicates occurring in rules. The static analysis of conditions evaluates their parsetree over 3-valued logic; truth values of leaves are computed by static analyses that are linear in the size of the respective policy sets.

5 Case study: attack-countermeasure trees

Peal+ and its tool *PEALT* can be used as an intermediate language into which domain-specific languages can be translated and analyzed. Such use has two benefits: analysis results can be certified, and *PEALT* may perform analyses that are not supported within the frameworks of those domain-specific languages. All scenarios found in this case study certified without refining any predicates.

We illustrate these benefits for attack-countermeasure trees (ACTs) [18] by means of an example ACT for a BGP reset of a session as discussed in [18]. The *PEALT* input code for this example would not really be meant for human consumption, as it would just be an intermediate syntax for facilitating analyses. Our translation extends the semantics of ACTs in that we may turn attack leaves, detection mechanisms, and mitigation mechanisms "on" or "off" – without compromising the computation of attack success probabilities, attack impact or attack cost. This, combined with the expressive conditions in *PEALT*, gives us richer analysis capabilities, discussed in detail below. The full *PEALT* code for this case study is built into the *PEALT* tool as an example case study.

Figure 9 shows the ACT taken from [18] where we merely annotated some of its nodes with policy names that we will use in our translation. This tree contains

AND and OR nodes as familiar from attack trees [12]. But it also contains three NOT nodes that all feed into parent AND nodes the possible effects of a pair of detection and mitigation mechanisms. Qualitatively, this means that such a pair of working detection and mitigation mechanisms will feed *false* into the parent AND node. The probabilistic interpretation in [18] is that both mechanisms have a probability of working, and so NOT nodes take as probability the complement of the product of these two probabilities of working mechanisms [18].

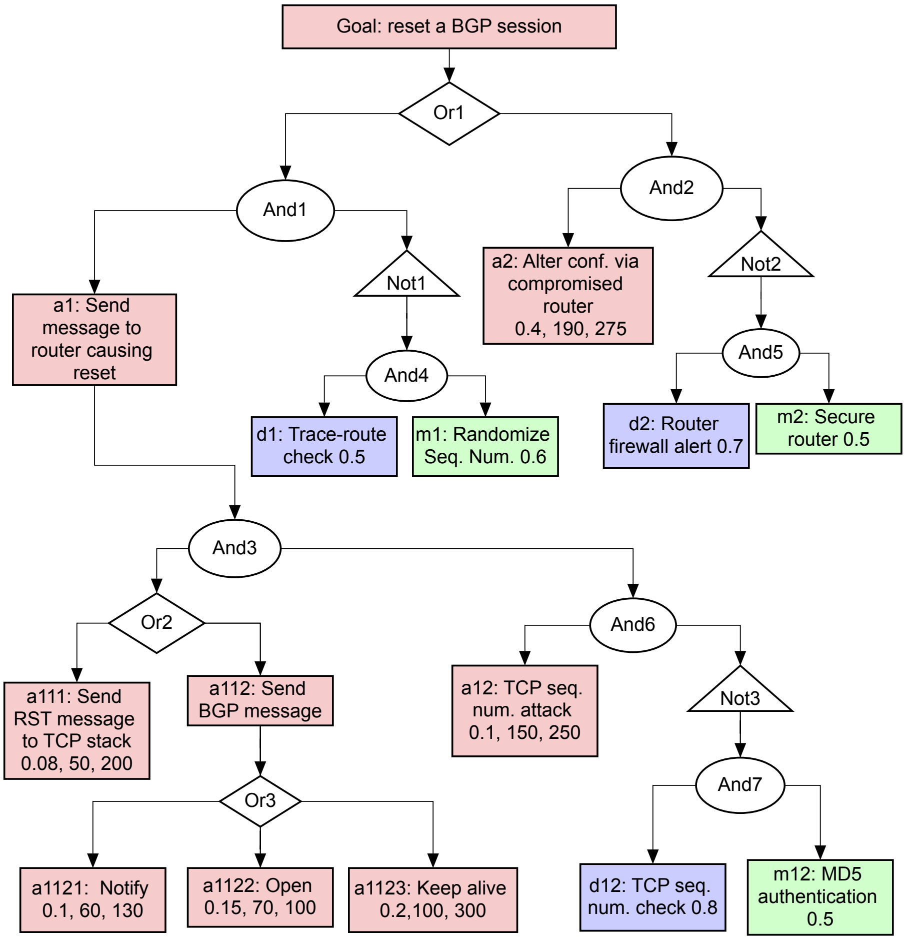

Goal: reset a BGP session
Or1
And1
And2
a1: Send message to router causing reset
Not1
a2: Alter conf. via compromised router 0.4, 190, 275
Not2
And4
And5
d1: Trace-route check 0.5
m1: Randomize Seq. Num. 0.6
d2: Router firewall alert 0.7
m2: Secure router 0.5
And3
Or2
And6
a111: Send RST message to TCP stack 0.08, 50, 200
a112: Send BGP message
a12: TCP seq. num. attack 0.1, 150, 250
Not3
Or3
And7
a1121: Notify 0.1, 60, 130
a1122: Open 0.15, 70, 100
a1123: Keep alive 0.2,100, 300
d12: TCP seq. num. check 0.8
m12: MD5 authentication 0.5

Fig. 9. ACT from [18] for reset of a BGP session, with detection/mitigation leaves' probability of working and attack leaves' success probability, cost, and impact (resp.)

The probability of attack success and impact cost are computed over the structure of the ACT [18], whereas attack cost is computed by first producing the set of all min-cuts (as used in fault tree analysis [2]) of the ACT [18]. This makes it hard to reason about the interaction of success probabilities, impact, and cost. Also, it faces scalability issues as the number of min-cuts may be exponential in the size of the ACT. We here want to demonstrate that the use of SMT solvers, facilitated with *Peal+* and *PEALT* as the intermediate language and tool, allows us to reason about such interactions and avoids the need to enumerate all min-cuts.

The declaration of policies for the probability of attack success, the result of policy `goal`, is shown in Figure 10. A predicate `True`, asserted to always be true, is used to compose results of children in the ACT. The probability at an OR node with n children x_i is $1-\prod_{k=1}^{n}(1-prob(x_k))$, and we expand this arithmetic term in stages using policy scores for stage composition, as seen for policy `or1`. The probability at an AND node with m children y_j is $\prod_{k=1}^{m} prob(x_k)$, and we similarly encode this arithmetic expression, as seen for policy `and1`.

For the encoding of attack leaves, their success probability is the score of a sole rule that captures that attack event. Since attack leaves are not under the scope of a NOT node, their default score is 0.0. The encoding of a NOT node is simply $1 - x$ where x is the result of its child AND node. For that AND node, the staged computation checks whether both detection and mitigation are present, in which case it computes the product of the probabilities of both mechanisms working; otherwise, it returns 0.0. This default score is sound as it makes the NOT node default to 1.0 which has no effect on its predecessors in the ACT (there is no NOT node in the scope of another NOT node). Thus this translation works for ACTs since they don't have nested NOT nodes.

In Figure 11, cost of attacks to an attacker and overall attack cost are specified. Default scores capture cost in the absence of attacks and so equal 0.0. In contrast to [18], overall cost is here the sum of all occurring, i.e. *true*, attacks since analyses ask whether attacks succeed within cost budgets and Z3 will search for such solutions by turning attack leaves "on" or "off" as desired. The specification of attack impact (now shown in this paper) reflects that the impact of an OR node is the maximum of the impact of all its children – modeling a worst-case scenario for the system [18]; and that the impact of an AND node is the sum of the impact of all its children. As in [18], NOT nodes don't contribute to impact of attack success, although it is noted in [18] that detection and mitigation mechanisms can reduce risk.

Finally, we may specify questions about this ACT in *Peal+*. Using basic conditions such as `549.0` $<$ `impact_overall` and binary conjunction, we express condition `c6` which asks whether the attack impact can be strictly above 549.0, the attack cost can be less than or equal to 440.0, and the probability of attack success can be strictly above 0.41199 – *all in the same scenario*. *PEALT* reports that this is possible in a scenario in which attacks `a1123`, `a2`, and `a12` occur (i.e. are *true*), as well as detection mechanisms `d1` and mitigation mechanism `m2`. The latter two may be unexpected. But in the scenario neither the mitigation mechanism `m1` of `d1` nor the detection mechanism `d2` of `m2` occur (i.e. are *false*). Therefore, none of the two respective NOT nodes contribute to the probability of attack success; and NOT nodes contribute neither to impact nor to cost.

```
goal = +((True or1_score)) default 1.0
or1 = +((True 1.0) (True -1.0*or1_aux_score)) default 1.0
or1_aux = *((True or1_aux1_score) (True or1_aux2_score)) default 1.0
or1_aux1 = +((True 1.0) (True -1.0*and1_score)) default 1.0
or1_aux2 = +((True 1.0) (True -1.0*and2_score)) default 1.0
and1 = *((True and3_score) (True not1_score)) default 1.0
and3 = *((True or2_score) (True and6_score)) default 1.0
or2 = +((True 1.0) (True -1.0*or2_aux_score)) default 1.0
or2_aux = *((True or2_aux1_score) (True or2_aux2_score)) default 1.0
or2_aux1 = +((True 1.0) (True -1.0*a111_score)) default 1.0
or2_aux2 = +((True 1.0) (True -1.0*or3_score)) default 1.0
a111 = +((sendRSTmessageToTCPStack 0.08)) default 0.0
or3 = +((True 1.0) (True -1.0*or3_aux_score)) default 1.0
or3_aux = *((True or3_aux1_score) (True or3_aux2_score)
                   (True or3_aux3_score)) default 1.0
or3_aux1 = +((True 1.0) (True -1.0*a1121_score)) default 1.0
or3_aux2 = +((True 1.0) (True -1.0*a1122_score)) default 1.0
or3_aux3 = +((True 1.0) (True -1.0*a1123_score)) default 1.0
a1121 = +((notify 0.1)) default 0.0
a1122 = +((open 0.15)) default 0.0
a1123 = +((keepAlive 0.2)) default 0.0
not1 = +((True 1.0) (True -1.0*and4_score)) default 1.0
and4 = +((traceRouteCheck and4_aux1_score)) default 0.0
and4_aux1 = +((randomizeSequenceNumbers and4_aux2_score)) default 0.0
and4_aux2 = *((True 0.5) (True 0.6)) default 1.0
```

Fig. 10. Policies that compute probability of attack success, even when certain attacks, detection mechanisms or mitigation mechanisms may be absent. Policies for sub-ACTs And2, And5, And6 and And7 are similar and not shown

```
cost_a111 = +((sendRSTmessageToTCPStack 50.0)) default 0.0
cost_a1121 = +((notify 60.0)) default 0.0
cost_a1122 = +((open 70.0)) default 0.0
cost_a1123 = +((keepAlive 100.0)) default 0.0
cost_a12 = +((TCPsequenceNumberAttack 150.0)) default 0.0
cost_a2 = +((alterConfigurationViaCompromisedRouter 190.0)) default 0.0
cost_overall = +((True cost_a111_score) (True cost_a1121_score)
                        (True cost_a1122_score) (True cost_a1123_score)
                        (True cost_a2_score) (True cost_a12_score)) default 0.0
```

Fig. 11. Computing cost of attack leaves and overall cost of occurring attacks

Threshold values chosen in condition `c6` are co-dependent: we can't decrease 440.0 by 1 or more, increase 549.0 by 1 or more, or increase 0.41199 by 0.00001 or more without making condition `c6` unsatisfiable. These values were determined by repeated analysis that adjusted values with bisection search using SAT and UNSAT results to drive the bisection method. If we add to condition `c6` a conjunct, saying that the detection/mitigation pair `d2` and `m2` also has to occur, $PEALT$ informs us that this is now impossible.

We can approximate maxima of security metrics, e.g., a measure of expected system damage $f(p,i,c) = p \cdot max(0, 2i - c)$ for attack success probability p, attack cost c, and attack impact i where we exploit that p, i, and c are expressed as policies. For example, $271.919999999999 < f(p,i,c)$ is satisfiable for the ACT in Figure 9 whereas $271.92 < f(p,i,c)$ is not. In $PEALT$, we have also implemented a bisection-based non-linear optimization for global maxima within specified accuracy – which can determine approximate maxima such as the one for the above security metric.

6 Discussion and Evaluation

We analyzed and tried to certify about $20,000$ random *Peal+* conditions with uncertainties but a few of these conditions failed to certify. We isolated the source of these failures to be an anomaly of the Z3 `push` command. With help of Arie Gurfinkel, Nikolaj Bjorner was able to attribute this to Z3 work item 108 (see `http://z3.codeplex.com/workitem/108`): if some constraints are non-linear, the use of `push` invokes a legacy solver that may report incorrect models for SAT outcomes. Since $PEALT$ won't use `push` when a sole analysis executes, we can eliminate this Z3 bug as the source of certification failure by turning off vacuity checking and commenting out all other analyses. We think $PEALT$ therefore strikes a good balance between performance (which the use of `push` on more than one analysis greatly improves) and correctness (since failed certifications are rare and mostly caused by typos as discussed next).

If a user declares a policy `p` but also writes `p` in a score instead of `p_score`, the SMT solver may find a real value for real variable `p` (implicitly declared in that rule!) and so $env(p)$ would have that value. If this is not the value one would statically infer for policy `p`, such aliasing will fail certification. Also, spelling mistakes in variable names may declare new variables that can result in inconclusive certification. Anecdotal evidence suggests that almost all failed or inconclusive certifications are results of such typos, which incidentally won't occur whenever $PEALT$ is used as intermediate language by code generators.

The certification process in $PEALT$ only works for scenarios (whose reported values for policy scores are ignored in certification), not for a claim that no scenario exists. We first focused our efforts on scenarios as they are likely to be more useful to specifiers, and since certification of non-existence of scenarios involves formal proofs extracted from back-ends (e.g. [3]), but general specifiers cannot be expected to understand complex proofs.

The scope of certification does not expand into section DOMAIN_SPECIFICS. For example, assume that a predicate occurs in no rule but is cast to a condition and declared in section DOMAIN_SPECIFICS, which also defines its meaning. Our certification will not inspect this definition of meaning as it is expressed outside of *Peal+* in an expressive logic. We did not find this to be limiting when writing and certifying *PEALT* models, but it means that certification is a relative notion in *PEALT*. On the other hand, it seems feasible to extend our 3-valued certification to cover DOMAIN_SPECIFICS as well for certain fragments of Z3's input language.

Our implementation of *Peal+* requires that policies be cast into policy sets (when needed), predicates be cast into conditions (when needed), and operators for policies, policy sets, and conditions be unary or binary (not n-ary). The latter is a good thing, since it means that all sub-conditions of conditions are explicitly declared and so subject to vacuity checking. *PEALT* does not check whether predicates within a policy occur more than once. The latter is an issue when two or more such occurrences have scores with uncertainty as this "binds" the non-deterministic choice made for these expressions to the same value. A variant of our BGP case study *with* uncertainties, built into tool *PEALT*, addresses this be using `True1`, `True2`, and so forth to disambiguate this.

PEALT has no explicit ability to model state spaces and their transition; one may see this as a weakness and opportunity for future work, or as a strength as it avoids state space explosions.

7 Related work

For model checking, Namjoshi developed deductive techniques in [15] that can independently verify the results of model checks for formulas of the modal mu-calculus and where these proofs can be extracted from an (instrumented) model checking run. For theorem proving, Gonthier [5] simplified the proof of the famous 4-color theorem, and proved it in the theorem prover Coq in such a manner that the proof itself could be certified as well. In [16], Necula devised a proof as a claim of certain program behavior, e.g. memory safety; it is efficient to verify the correctness of the proof (though producing the proof may have been hard) and one can check whether its claim is consistent with one's own security policy.

Jha et al. [8] use model checking to automatically generate attack graphs with nodes representing network states, develop techniques for choosing minimal number of security measures and for trading off attack likelihood and attack probability. Attack graphs that express dependencies of vulnerabilities instead, such as those of Albanese et al. [1], have more scalable analyses than state-based ones. Attack graph models in the literature appear to have a fixed model signature, whereas *PEALT* can extend modeling domains as and when needed.

Peal+ extends *Peal* [4] and the *PEALT* tool over its version in [7]: *PEALT* now supports the richer language *Peal+*, automated vacuity checking of all declared analyses, the automated certification of all scenarios generated by Z3 for analyses, and the partial evaluation of policies over scenarios so that users can comprehend scenario information directly on relevant policies.

In [6], we sketched *Peal+* and illustrated it with mock-up syntax for a "score card" model very similar to that from Figure 4. Although that paper discussed usability issues, it focussed on the design of *Peal+* and did not cover usability issues of a supporting tool and its user feedback.

8 Conclusions and future work

We presented a domain-specific language *Peal+* in which the interaction of concepts that inform security and threat analysis can be formally expressed and analyzed. We reported its implementation in the $PEALT$ tool that statically analyzes such conditions with two principal aims: to determine whether specified conditions meet expectations of how security-related concepts influence decision making; and to validate that the expectations that users have do not have unintended consequences when expressed and enforced in such conditions.

$PEALT$ reflects the methodology of *auto-interactive verification* (see Fig. 1). This means users can rely on automated verification tools that provide easily comprehended feedback which may trigger subsequent modeling and automated verification. And this process would be repeated until users are satisfied to have captured conditions as desired. This paper realized this methodology via a language *Peal+*, its implementation in the $PEALT$ tool, and use of the SMT solver Z3 as the back-end for automated reasoning and scenario generation.

We illustrated the utility of *Peal+* and these support mechanisms by first discussing a Car Rental Risks example and then attack-countermeasure trees. We showed how ACTs can be translated into *Peal+* so that we can reason about interaction of the probability of attack success, attack cost, and attack impact whilst at the same time allow the model to turn attack, detection, and mitigation leaves "on" or "off" at will. Therefore, our ACTs actually represent an entire set of ACTs and we can verify invariants of such interaction over that set of ACTs.

We created support for validating scenarios computed for conditions expressed in *Peal+*: an independent certification of the correctness of scenarios with respect to the domain and policies in which they should be interpreted. We stress that our certification is agnostic to the manner in which code for analysis in back-ends is generated (since certification operates on *Peal+* expressions directly) and agnostic to the choice of back-end (apart from an interface for the scenario to be certified and for variables modelling uncertainty). $PEALT$ partially evaluates all policies that certification seems to rely upon, with respect to the certified scenario and provides this as auxiliary feedback, so that modelers may more easily assess the impact of policies certified in possibly refined scenarios.

We could extend *Peal+* with judicious support for integer variables (a potential performance bottleneck for SMT solvers). We also mean to develop auxiliary tools that can translate other threat modeling formalisms into $PEALT$ for richer analysis, as illustrated for ACTs in this paper. Finally, we mean to research how we can extend *Peal+*, $PEALT$, and our certification to state transitions and to conditions that analyse state changes through operators of temporal logic.

Open access: We refer to URL `http://www.doc.ic.ac.uk/~hk2109/PEALT/` for the latest version of *PEALT* and installation instructions. Please consult `https://bitbucket.org/jimhkuo/pealapp-lift` for the Scala source code.

References

1. Albanese, M., Jajodia, S., Noel, S.: Time-efficient and cost-effective network hardening using attack graphs. In: Proc. of the 42nd Ann. IEEE/IFIP Int'l Conf. on Dependable Systems and Networks (DSN). pp. 1–12. IEEE Computer Society (2012)
2. Bedford, T., Cooke, R.: Probabilistic Risk Analysis: Foundations and Methods. Cambridge University Press (2001)
3. Böhme, S., Weber, T.: Fast LCF-style proof reconstruction for Z3. In: ITP. pp. 179–194 (2010)
4. Crampton, J., Huth, M., Morisset, C.: Policy-based access control from numerical evidence. Tech. Rep. 2013/6, Imperial College London, Department of Computing (October 2013), ISSN 1469-4166 (Print), ISSN 1469-4174 (Online)
5. Gonthier, G.: The four colour theorem: Engineering of a formal proof. In: 8th Asian Symp. on Computer Mathematics. p. 333. LNCS 5081, Springer (2007)
6. Huth, M., Kuo, J.H.P.: On designing usable policy languages for declarative trust aggregation. In: HCI (24). pp. 45–56 (2014)
7. Huth, M., Kuo, J.H.P.: PEALT: An automated reasoning tool for numerical aggregation of trust evidence. In: Proc. of TACAS 2014. Lecture Notes in Computer Science, vol. 8413, pp. 109–123. Springer (2014)
8. Jha, S., Sheyner, O., Wing, J.: Two formal analys s of attack graphs. In: Proc. of 15th IEEE Workshop on Comp. Sec. Found. pp. 49–63. IEEE Comp. Soc. (2002)
9. Kleene, S.C.: Introduction to Metamathematics. North Holland (1952)
10. Korzhyk, D., Yin, Z., Kiekintveld, C., Conitzer, V., Tambe, M.: Stackelberg vs. nash in security games: An extended investigation of interchangeability, equivalence, and uniqueness. J. Artif. Int. Res. 41(2), 297–327 (May 2011)
11. Kupferman, O., Vardi, M.Y.: Vacuity detection in temporal model checking. In: 10th IFIP WG 10.5 Advanced Research Working Conference (CHARME'99). pp. 82–96. LNCS 1703, Springer (1999)
12. Mauw, S., Oostdijk, M.: Foundations of attack trees. In: Proc. of 8th Int'l Conf. on Information Security and Cryptology. pp. 186–198. LNCS 3935, Springer (2006)
13. Moore, A., Ellison, R., Linger, R.: Attack modeling for information security and survivability. Tech. Rep. Technical Note CMU/SEI-2001-TN-00, Software Engineering Institute, Carnegie Mellon University (2000)
14. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS. pp. 337–340 (2008)
15. Namjoshi, K.S.: Certifying model checkers. In: 13th Int'l Conf. on Computer Aided Verification. pp. 2–13. LNCS 2102, Springer (2001)
16. Necula, G.C.: Proof-carrying code. In: Proc. of POPL'97. pp. 106–119. ACM Press (1997)
17. Pavlidis, N.G., Tasoulis, D.K., Adams, N.M., Hand, D.J.: Adaptive consumer credit classification. Journal of the Operational Research Society 63(12), 1645–1654 (2012)
18. Roy, A., Kim, D.S., Trivedi, K.S.: Attack countermeasure trees (ACT): towards unifying the constructs of attack and defense trees. Security and Communication Networks 5(8), 929–943 (2012)
19. Schneier, B.: Secrets and Lies: Digital Security in a Networked World. John Wiley and Sons Inc. (2000)
20. Wood, M.: Flaw calls for altering passwords, experts say. The New York Times (Technology) (9 April 2014)

A Hierarchical Memory Management for a Load-Balancing Stream Processing Middleware on Tiled Architectures [*]

Nilesh Karavadara, Michael Zolda, Vu Thien Nga Nguyen, Raimund Kirner

University of Hertfordshire
n.karavadara, m.zolda, v.t.nguyen, r.kirner@herts.ac.uk

Abstract. Tiled many-core computer architectures are becoming increasingly popular, providing a viable solution to the complexity of resource management in parallel processors. One of the critical challenges in programming such tiled many-core architectures is the efficient use of available resources.

In this paper we present a hierarchical memory management approach for tiled many-core processors. This memory management approach is capable to provide shared memory across multiple OS instances running on different cores. This memory management approach made it possible for us to port LPEL, a dynamic load-balancing middleware for stream processing applications, to the Intel Single-Chip Cloud Computer (SCC), a research processor that shares many similarities with other tiled architectures. This is the first execution middleware running on the SCC that provides dynamic load balancing. An evaluation shows that our framework performs better than an MPI-based implementation.

1 Introduction

The demand for compute power is exceeding the supply [20]. Until recently, increasing the processor clock speed to meet the demand of performance had worked well, but heat and interference caused by increased clock speeds and shrinking size of transistors are starting to limit processor designs [5, 4].

The current strategy is to use several simple and power efficient cores [27] instead of a single complex and power-hungry core. Processors such as the Intel Xeon Phi [7], ARM (Cortex A7 & A15) [12], Tilera (Tile-Mx & Tile-Gx) [10, 25] and Kalray MPPA 256 [18] reflect a trend towards tiled architectures.

Parallel architectures with multi-core tends to be more energy-efficient than an equivalent single-core processor. However, programming multi-core processors is more complex and requires the refactoring of algorithms for concurrency. The identification and exposition of concurrency is not enough to exploit a high fraction of the potential computing power made available by a parallel platform. The scheduling of dynamic

[*] This work has been supported by the Material Transfer Agreement 2010-2013 for the Intel SCC Research Processor “Light-weight Parallel Execution Layer for Stream Processing Networks on Distributed Memory Architectures” and the FP7 ARTEMIS-JU research project “ConstRaint and Application driven Framework for Tailoring Embedded Real-time Systems” (CRAFTERS) under contract no 295371.

workload on these tiled platforms increases the difficulty in utilising available resources. Automatic load-balancing reduces the waste of resources and relieves the programmer.

Dataflow programming [17] is a particularly promising model for concurrent programming. Data flow languages expose concurrency directly through explicit modelling of data dependencies, in contrast to most traditional programming languages which are centered around control flow. *Coordination languages* [8] allow software engineers to build parallel applications from sequential building blocks. *Stream processing* [28] is a parallel execution model that is well-suited for architectures with multiple computational elements that are connected by a network. Put together, these mechanisms afford a powerful software development approach for multi-cores [14, 13, 24].

In this paper we introduce a hierarchical memory management approach for tiled many-core processors that provides shared memory across multiple OS instances. Like the Hoard memory manager [3] and scalloc [1] our memory management approach uses local memory pools per core. In contrast to Hoard and scalloc, ours also works with separate OS instances. Based on that memory manager we ported the *Light-weight Parallel Execution Layer* (LPEL) [24, 21] to the Intel SCC research processor, making it the first execution middleware running on the SCC with dynamic load balancing.

Sections 2.1 and 2.2 of this paper offer a brief review of the SCC tiled architecture. Sections 2.3 and 2.4 provide a short review of the stream programming paradigm and the stream execution model. Section 3 describes how our middleware maps streaming network on the SCC tiled architecture. In Section 4 we study the issue of caching and compare our middleware with an existing MPI [2] implementation. We discuss relevant related work in Section 5 and conclude with Section 6.

2 Preliminaries

2.1 The SCC Architecture

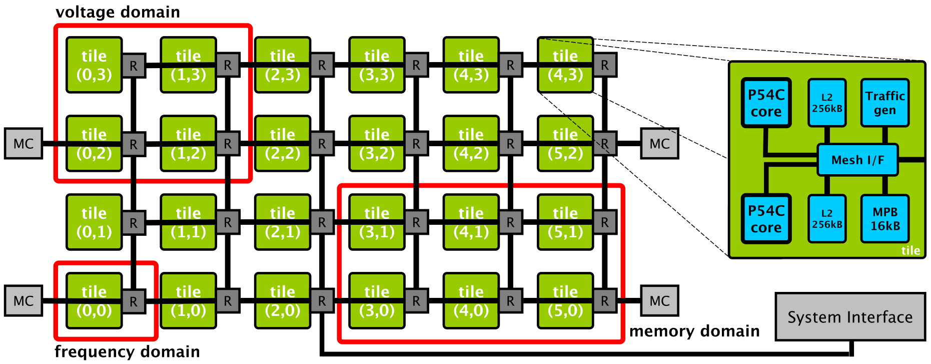


Fig. 1: SCC Top-Level Tile Architecture

The SCC [15, 26, 16] is an experimental tiled multi-core processor created by INTEL. The processor consists of 24 tiles in a 4x6 grid, connected by a high bandwidth, low latency, on-die 2D mesh network, resembling a cluster on a single chip, as shown in Fig. 1. Each tile hosts two modified P54C processor cores that support x86 compilers and operating systems. Each core has 32 kB L1 and 256 kB L2 cache. Furthermore, each tile has a 16 kB block of SRAM called Message Passing Buffer (MPB), which is physically distributed, but logically shared. Each tile connects to the mesh network via a router.There is a Voltage Regulator Controller (VRC) to let programs dynamically manage the voltage and the frequency of cores, and four on-die Memory Controllers (MC), which support a total of 16 to 64 GiB off-die DRAM. Only one atomic test-and-set register and two atomic counter registers are available per core via the system interface, which is a limiting factor for efficient synchronisation.

The SCC is a research processor, but many of its features are found in commercial processors, for example in Tilera TILEPro series: In Tilera's architecture, cores are also organised as 2D grid of tiles connected to a mesh via on-tile routers, and each core is capable to run an OS instance. Each tile has 16 kB L1 instruction cache, 8 kB L1 data cache, and a 64 kB combined L2 cache. In contrast to SCC's mesh network, the TILEPro has six independent networks to route traffic to different destination, i.e., tile, tile caches, external memory, and IO Controllers. The Special Purpose Registers (SPRs) are nearly identical to SCC's control register buffer (CRB). There are three memory modes: In default mode the hardware maintains cache coherence, but it does not do so in non-coherent mode, and in non-cacheable mode all the data blocks are not cached at any level. Dynamic Distributed Cached Shared Memory (DDC) on TILEPro serves same purpose as MPB on SCC [30].

2.2 The SCC Memory Architecture

The SCC offers three address spaces:

- A private off-chip address space in DRAM for each core. This memory is cache-coherent with an individual core's L1 and L2 caches.
- A shared off-chip address space in DRAM. This memory can optionally be configured as cached, but ensuring cache-coherence is the programmer's responsibility.
- The MBP, a physically distributed, logically shared address space in SRAM.

Tiles are organised in four *memory domains* of six tiles each. Each memory domain maps to a particular MC. Private memory is accessed through the assigned MC, shared memory can be accessed through any of the four MCs.

Each core has its own 256-entry Lookup Table (LUT) to translate 32-bit core addresses to a 46-bit system addresses. Each core can address 4 GiB of physical memory, even though the SCC supports up-to 64 GiB in total. The LUTs provide a mechanism to translate 32-bit physical core address to 34-bit physical system address. The upper 8 bits of a physical core address index a LUT entry, which contains 22 bits, of which the upper 12 are routing information. The lower 10 bits are prepended to the lower 24 bits of the core address, resulting in the 34-bit memory address. The LUTs are loaded with default values at boot time, but it is possible to change them dynamically.

The interaction of the memory with caches depends on its mapping. The part of the DRAM that is mapped as a shared region between cores can be configured to be cached or uncached. If memory is configured as cached, read/write accesses go through the L1 and L2 caches and manual flushing of the L2 cache is required to commit data to main memory. The SCC does not provide cache coherence, hence concurrent accesses to shared data may cause memory consistency issues in cached mode. If the memory is configured as uncached, read/write accesses go directly to main memory (DRAM).

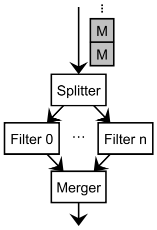


Fig. 2: Image Filter

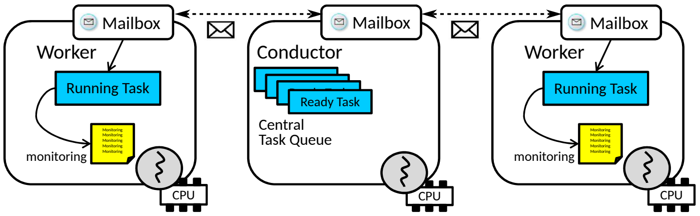


Fig. 3: LPEL Execution Layer

There is a tag for data in the MPB called Message Passing Buffer Type (MPBT) that identifies L1 cache lines. Tagged data bypasses the L2 cache and goes directly to the L1 cache in the case of reads. Write operations to tagged memory are stored in the Write Combine Buffer (WCB) until an entire cache line is filled or a write access to a different cache line happens. Intel has also extended the Instruction Set Architecture (ISA) with an instruction to invalidate all tagged cache lines in L1. Accessing invalidated L1 cache lines forces an update of the L1 cache lines with the data in the shared memory.

2.3 Stream Programming

In stream programming, a program is structured as a set of computation processes called nodes and a set of directed communication channels between them called streams. Stream programs can be viewed as a graphs whose vertices are nodes and whose edges are streams. Streaming data is presented as an infinite sequence of messages. Examples of stream programming can be found in [6, 14, 29].

Fig. 2 shows an example of an S-Net [14] program — an image filter. The *Splitter* node consumes an image and splits it into sub-images. The number of sub-images varies depending on the size of the original image. The sub-images are sent to different branches where *Filter* nodes perform the actual filtering operation. The processed sub-images are sent to the *Merger* node, which combines them into a complete image.

2.4 LPEL - A Stream Execution Layer with Efficient Scheduling

Our streaming middleware includes two layers: a *runtime system* (RTS) and an *execution layer*. At the RTS layer, each stream is represented as a FIFO message buffer and each node of the stream program is transformed into a task. A task is an iterating process that reads messages from its input streams, performs the associated node's computations, and writes output messages to its output streams. The role of the RTS is to enforce the semantic of stream programs, i.e., to ensure that each task reads from and writes to its appropriate streams. The execution layer below the RTS provides primitives for task and stream management and a scheduler that distributes tasks to cores.

The Light-weight Parallel Execution Layer (LPEL) [24] is an execution layer providing user-space threading and communication mechanisms for stream programs on shared memory platforms. It provides functions for creating, reading, writing and modifying streams and a task component to create a wrapper around each node before sending it to the scheduler.

LPEL offers two different schedulers: DS-LPEL uses a global mapper to allocate tasks to cores and a local scheduler for each core. The local scheduling policy is round-robin, whereas the mapper uses either a round-robin policy or a static mapping [24].

HRC-LPEL follows a centralised approach of automatic load balancing [21], where one *conductor* core is dedicated to manage the set of ready tasks; cf. Fig. 3.

3 LPEL on the SCC

To obtain good performance in terms of throughput and latency, the HRC-LPEL scheduler uses the notion of data demands on streams to derive the task priority that is used to decide which task will be executed on available core. We deployed HRC-LPEL on the SCC, as it has been shown to be more efficient than the DS-LPEL [21]. Furthermore, it better suits our future plan to extend the scheduler with power management features.

HRC-LPEL requires shared memory to efficiently move tasks and their associated states between the workers, however the SCC is by default a distributed memory platform: Each core runs separate OS instance.

Although the SCC offers a fast network between the cores and on-chip shared memory (MPB), the default configuration does not offer enough shared memory. The limited number of hardware locks also makes it difficult to deploy HRC-LPEL on the SCC, as we need at least one lock per stream. Software mechanisms like mutexes from the POSIX thread library are not safe to use on distributed memory platforms.

For all these reasons we re-configure the SCC as a shared memory platform.

3.1 Shared Memory Creation

We use the LUT entries described in Section 2.2 to configure the SCC such that it behaves as a shared memory platform. There are 256 LUT entries, of which 0–40 are used by OS that is running on the core. Entries 192–255 are mapped to the MPBs and configuration registers. This leaves LUT entries 41–191 unused.

To create shared memory we improve upon technique used in the RCCE [31] library. In RCCE, 4 LUT entries are mapped to same physical address-space range on all the cores. As each LUT entry points to a 16 MiB segment of physical memory, this mapping provides 64 MiB of memory that is shared between all cores. There are two problems with this approach: Firstly, 64 MiB are not enough to deploy HRC-LPEL and run some real-world application. The lack of shared memory can be addressed using the remaining LUT entries. As each entry points to a 16 MiB chunk of memory this provides us approx 2.5 GiB of shared memory. As mentioned before, 41 of the 256 entries point to physical memory needed by the individual OS instances. To obtain more shared memory we disable 4 of the 48 cores and use entries from those cores to populate unused entries of LUT.

The second problem is that with memory mapped as described in RCCE you get a globally visible shared memory, but the virtual address range is not the same for all the cores. In this case, pointers are not globally valid. Using offsets from the beginning of the address-space instead would introduce an additional overhead. We solve the problem by mapping the address-space to same virtual address range on all cores.

Calling standard malloc will allocate space in private rather than shared region of memory. We have written our own malloc and free functions that are based on K&R malloc and free [19] to address the issue.

3.2 Shared Memory Management

By using LUT entries to create shared memory, all cores get the same view on the memory. There are multiple ways we can allocate this shared memory to cores.

The first approach is to have a global allocator that allocates memory to each core as and when requested. As shared memory is global each core has to grab the lock, allocate memory and release lock. The lock is necessary as we do not want meta-data within the memory allocator to be corrupted due to simultaneous accesses by multiple cores. This creates unnecessary contention and adversely impacts the performance.

In the second approach, the global memory is divided into chunks of equal size and then each core can locally manage its chunk. The problem with this approach is that not all tasks need the same amount of memory. If we distribute equally-sized chunks of memory to all the cores, we waste resources.

To alleviate this problem we can fuse the first and second approach to create a hybrid allocator. We can have a global allocator that allocates chunks of memory as and when required by participating cores and then cores allocate memory locally from these chunks. When cores do not have enough memory to fulfil the next request for memory allocation, it will request another chunk from the global allocator.

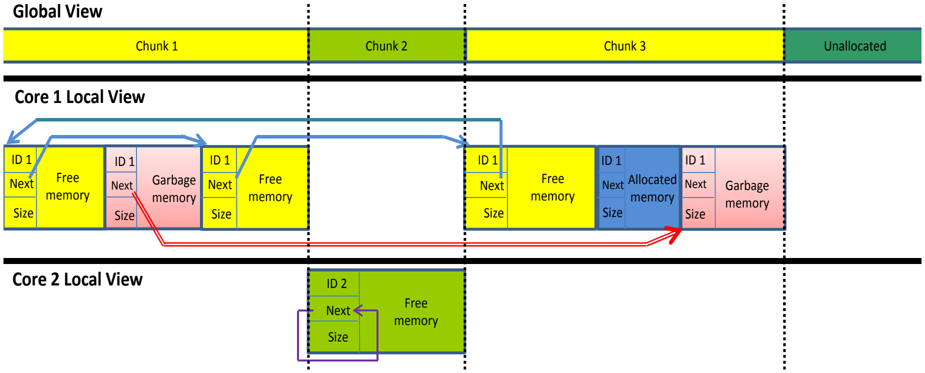


Fig. 4: Shared memory layout

Figure 4 depicts the view of shared memory from perspective of different cores. When a core makes a request to the global allocator it receives a chunk of memory. The chunks that are managed by a core may not be contiguous. Thus the core keeps its free storage as a circular list of free blocks, e.g., core 1 manages chunk 1 and 3. If we consider core 1's view at the chunks then both chunks are divided into small multiple blocks. The local allocator manages two lists; first to keep track of free storage, known as *free list*. Second list is *garbage list* to keep track of garbage storage that needs to be added back to free list. Each block contains a header indicating its size, a pointer to the next block, and an owner id, followed by the actual memory.

Algorithm 1 allocates n bytes of memory from local shared memory. The local allocator scc_malloc_local uses a "first fit" algorithm [19] that is not thread-safe. It is therefore protected by a lock (lines 1–3). A return value of NULL means there is no block available from where core can allocate required memory (line 4). The core then requests a new chunk of memory from the global allocator (line 5). In order to ease the contention on global allocator the core always requests m bytes where $m > n$. To make this newly allocated memory chunk available to the requesting core, we call scc_free_local on it so that it gets added to the free list (line 7).

The function scc_malloc_local is called again to allocate memory. If the return value is still NULL that means there is not enough memory available and an error is returned (lines 8–13). All the calls to scc_malloc_local and scc_free_local are protected by a mutex from the POSIX thread library to make them thread-safe.

Algorithm 2 is a simple allocator that keeps track of the size of the global shared memory and the starting point of this memory as meta-data. When a request for memory allocation is made by a core, the global allocator performs three steps: First it checks, if there is enough memory to allocate. If so, it continues to the next step—otherwise it returns an error. Next it uses a lock implemented by using a test-and-set register to avoid any corruption of meta-data. Finally it allocates the required memory block and adjusts the size and starting point of the global shared memory before releasing the lock. This hierarchical malloc means we will also need a hierarchical free.

Algorithm 1 *scc_malloc_local* to allocate n bytes from local shared memory

```
 1: $mutex\_lock()$
 2: memptr ← scc_malloc_local(n)      ▷ K&R malloc
 3: $mutex\_unlock()$

 4: if memprt = NULL then
 5:     chunk ← scc_malloc_global(m)
 6:     $mutex\_lock()$
 7:     scc_free_local(chunk)
 8:     memptr ← scc_malloc_local(n)
 9:     $mutex\_unlock()$

10: if memprt = NULL then
11:     Not enough memory available, return Error
12: else
13:     Return memptr
```

Algorithm 2 *scc_malloc_global* to allocate m bytes from global shared memory

```
Require: $m \leq available\_global\_memory$
 1: $tas\_lock()$
 2: chunk ← $m$ bytes from global memory
 3: $tas\_unlock()$
 4: Return chunk
```

Algorithm 3 *scc_free_local* to free memory pointed by p

```
 1: if not (shmStart $< p <$ shmEnd) then
 2:     standard_free(p)      ▷ $p$ points to private memory
 3:     return
 4: if owner id = core id then
 5:     $mutex\_lock()$
 6:     scc_free_local($p$)      ▷ K&R free
 7:     $mutex\_unlock()$
 8: else
 9:     $acr\_lock()$
10:     add $p$ to garbage list of core with owner id
11:     $acr\_unlock()$
```

Algorithm 4 *scc_free_garbage* to free memory from garbage list

```
 1: $acr\_lock()$
 2: glfirst ← gl      ▷ copy the garbage list gl
 3: gl ← NULL      ▷ empty the garbage list
 4: $acr\_unlock()$

 5: $mutex\_lock()$
 6: while $glfirst \neq NULL$ do
 7:     glnext ← block after glfirst
 8:     scc_free_local($glfirst$)      ▷ K&R free
 9:     glfirst ← glnext
10: $mutex\_unlock()$
```

Algorithms 3 and 4 free the shared memory that was allocated with our own allocator function scc_malloc_local. We know the range of the global shared memory region and can check if the memory that is being freed is within this shared region or not (line 1). If the memory pointed by p was allocated in private region using standard malloc, then we need to free it using standard free (line 2). If it was allocated in shared region then owner id from memory block and core id are compared (line 4). If the owner id and core id are the same, then we call function scc_free_local, which is the standard free function corresponding to scc_malloc_local [19]. As mentioned earlier calls to scc_free_local are protected by a lock to ensure thread-safe operation (lines 5–7).

Each core maintains a garbage list of blocks to be freed. Access to this garbage list is protected by locks. In case of id mismatch, the core will add the block to the garbage list of the core that allocated the block (lines 9–11).

Algorithm 4 frees the memory blocks that were added to its garbage list by some other cores. This algorithm is executed periodically during scheduling cycle.

Here we use two different locks. The first lock is to protect the garbage list from being corrupted due to the concurrent access by other cores (lines 1,4). We use atomic counter registers of the SCC to implement this lock. The second lock is local to the core to ensure thread-safe operation of scc_malloc_local and scc_free_local by using a pthread mutex (lines 5,10). Once the garbage list is copied and the original list is emptied, we can release the lock so that other cores can start adding memory block to be freed (lines 2,3). Then we loop through copied list and add blocks to free list (lines 6–9) by calling function scc_free_local (line 8).

3.3 Conductor/Worker Initialisation

When deploying the HRC-LPEL scheduler on the SCC, it makes sense to create exactly as many workers as there are cores, as the cores of SCC are single-threaded. As there is no shared memory at the beginning, we can not just create conductor/workers on a single core and then distribute them amongst participating cores. For this purpose, when the execution of a program starts, a configuration file is used to decide which core will be the conductor based on the physical core id.

As mentioned in Section 3.1, to create a truly globally shared memory, all cores have to map part of program's address-space to same virtual address range. At the beginning there is no shared memory, apart from the MPB. Meta-data, including a flag necessary to establish communication between cores is located in a pre-defined location in MPB.

If a core is a conductor, it starts by initialising the shared memory, tasks, streams and the static parts of streaming network. If core is a worker, it will busy-wait on a flag located in MPB. Once the conductor has mapped the LUT entries and created the shared memory, it places the relevant LUT configuration in the MPB and sets the flag. Once the flag is set the worker cores configure their LUTs to map shared memory to same virtual address range as conductor.

HRC-LPEL uses mailboxes to facilitate communication between conductor and workers. Each mailbox is protected by a lock to ensure that no messages are lost. Once the mailboxes are setup, the workers request tasks to execute from the conductor and the conductor will fulfil these requests based on demand and task priority. When there are no more messages to be processed, the conductor sends a termination message to all the workers via mailbox.

3.4 Synchronization primitives

HRC-LPEL requires a number of means to synchronise at different points.

- when initialising, the conductor/workers need a shared flag
- the meta-data of the global shared memory may be accessed by conductor/workers concurrently
- meta-data of the local shared memory needs to be protected against concurrent access by multiple threads
- the mailbox is an example of producer/consumer, where messages are added/removed from queue. This queue needs to be protected against concurrent access to ensure messages are not lost
- streams are used to transfer data/messages between tasks. Streams are implemented as FIFO buffers, and these buffers need protection to ensure integrity of data (during reading/writing to the stream).

We already use all the hardware registers provided by SCC for synchronisation. The MPB is used to store the shared flag. We use the test-and-set registers to implement locks that protect the meta-data of the global shared memory. We use the atomic counter registers to implement locks to protect the garbage list and the mailboxes.

We still need synchronisation primitives to protect the streams and for allocating core local shared memory. For this purpose we use POSIX (pthread) mutexes. The SCC runs an OS instance on each core, so we create mutexes with the process shared attribute set. When different worker threads try to access the same mutex, it will be seen as it was accessed by different processes.

4 Experiments

We evaluate the efficiency of HRC-LPEL with dynamic load balancing on the SCC and compare it to DS-LPEL with manual load balancing. In the latter each core has its local round-robin scheduler, and the cores communicate via MPI. We also evaluate the scalability of HRC-LPEL for varying numbers of cores.

4.1 Experimental Setup

In our experiments we used a default sccKit 1.4.2 configuration, with the cores running SCCLinux at 533MHz, and memory and mesh running at 800MHz. We used the SCCLinux driver for memory mapping.

We used four benchmarks implemented using the S-Net coordination language [14]:

- DES: Encrypts data using DES. This benchmark performs computationally intensive operations on relatively small chunks of data of 2 kB.
- FFT: Calculates a fast fourier transform. This benchmark performs computationally less intensive operation on relatively large chunks of data of 64 kB.
- HIST: Calculates histograms of images. This benchmark performs computationally intensive operations on relatively large chunks of data of 127 kB.
- FILT: Applies a series of filters on images. This benchmark performs computationally intensive operations on relatively large chunks of data of 127 kB.

Each benchmark contains a pipeline performing the application's main function. To increase the level of concurrency, S-Net provides parallel replication to create multiple instances of the pipeline.

We used 4 out of the 48 cores as donors for shared memory, and 4 further cores to model an external source/producer and sink/consumer for stream programs. Since we need at least one conductor and one worker, our baseline is 2 cores.

The SCC does not provide cache coherency and offers no direct control over cache flushing, so we have to ensure consistency when using the cache. We use two variants of HRC-LPEL: In DLB only the task stack—consisting of non-shared data—is cached, whereas in NDLB we do not use caching. For DS-LPEL with manual load balancing MPI is used and memory is not shared, so we can make full use of caching. In this approach, which we denote MLB, each benchmark is mapped to achieve the best load balance, i.e., each instance of the pipeline is mapped on a separate core.

The first core is special: Besides processing messages, it is also responsible for receiving input messages from the environment, distributing messages to the other cores and collecting them, and sending them out to the environment. The MPI communications occur only between the first and all other cores. To ensure the message order, MPI must be used in blocking mode.

4.2 Experimental Results

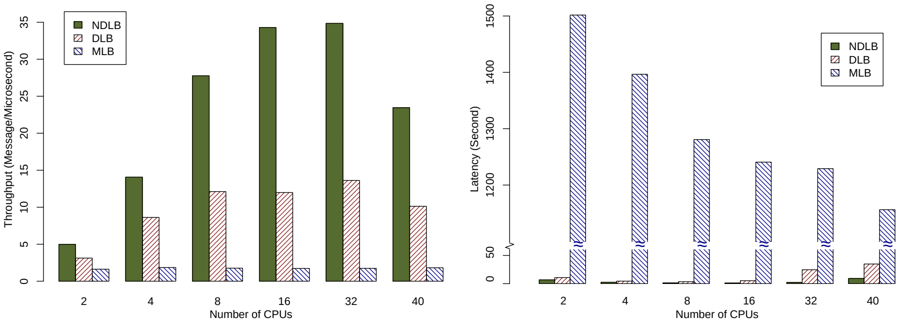


Fig. 5: Performance of FFT on NDLB, DLB and MLB

Fig. 5 shows the maximum throughput and minimum latency of the FFT benchmark. NDLB outperforms DLB by a factor of at least 1.5 for both, throughput and latency, even though caching is disable in NDLB. Since the SCC is configured as a shared memory platform, the caches need to be flushed to ensure data integrity among cores. This causes a significant overhead that caching cannot compensate.

MLB has the lowest throughput, because the communication performance of MPI is inferior to direct memory access. The maximum communication bandwidth between 2 cores is around 2.78 MiB/s for MPI. Transferring 64 kB between 2 cores takes more than 22 ms via MPI but only 15 ms via direct memory access. With 2 cores the throughput for MLB is smaller than for DLB and NDLB, and for more cores the MPI bandwidth is shared. MLB requires one core to communicate with all other cores, sending input messages and receiving output messages. Due to similar load on the cores, this communication is likely to coincide. MPI introduces a (de)serialising and (un)packing overhead and operates in blocking mode and this forces each core to wait while sending messages via the MPI interface. As a result the MLB throughput for MLB can be 7 times smaller than for DLB and 20 times smaller than for NDLB, as shown in Fig. 5.

MLB has a higher latency than NDLB and DLB. Besides the beforementioned reasons, the HRC-LPEL scheduler affords control over the consumption rate of input mes-

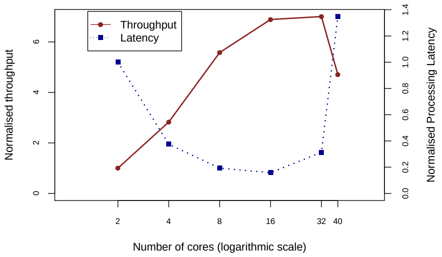


Fig. 6: Scalability of FFT on NDLB

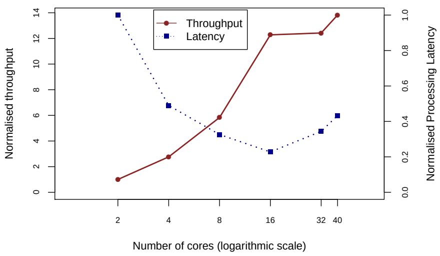


Fig. 7: Scalability of DES on NDLB

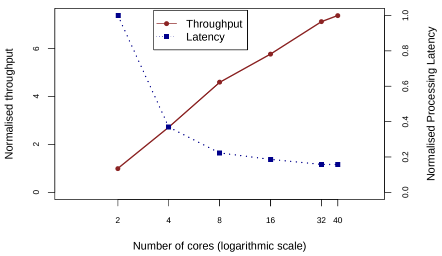


Fig. 8: Scalability of Histogram on NDLB

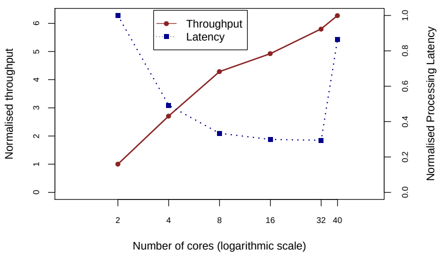


Fig. 9: Scalability of Filter on NDLB

sages to optimise latency [21]. MLB lacks this feature and allows the program to consume input messages even when it is overloaded and unable to process them. The latency for MLB can be 370 and 900 times higher than for DLB and NDLB, respectively.

Fig. 6 shows how NDLB scales for the FFT benchmark. From 2 to 16 cores the throughput scales roughly linearly, but more cores imply more memory accesses. Memory is managed by 4 memory controllers and extensive access can cause contention. Therefore throughput does not scale well between 32 and 40 cores. Although FFT operates on a sizeable amount of data (64 kB), the computation time is relatively small. On average each task takes 65 ms to process a message, so each core must access a large amount of data frequently.

In contrast, DES requires extensive computation on a small amount of data. Each input message is 2 kB and each task takes 194 ms on average to process a message. For this reason, the throughput of DES scales better, as shown in Fig. 7.

The latency depends on the immanent concurrency level of the stream program. Increasing the number of cores takes advantage of the concurrency within the stream program and helps to reduce the latency. However, more cores also imply higher communication costs, as tasks are spread among cores. Figs. 6 and 7 show that the latency decreases when we increase the number of core up to 16. For 32 and 40 cores the communication overhead surpasses the benefit of concurrency. The latency of DES and FFT therefore does not scale well for 32 or 40 cores.

Fig. 8 and Fig. 9 shows throughput and latency for HIST and FILT benchmark respectively. In contrast to DES and FFT here we can see roughly linear scaling in throughput from 2 cores all the way to 40 cores. This was expected, as HIST and FILT are computationally more intensive than DES and FFT. For HIST the latency continues to decrease up to 40 cores. In contrast, a decrease in latency can be observed for FILT for up to 32 cores, after which it rises sharply. One reason can be the higher number of message queuing at merge point in the stream network, which can be a bottleneck.

Table 1 shows the minimal and maximal execution time for each task in the benchmarks. Some of these tasks have multiple instances occurring in the separate parallel pipelines created by S-Net. We can see that all benchmarks show a considerable variation in execution times of tasks. This can be attributed to high work-load imbalance which depends highly on input messages. These numbers underline the need for a load balancing scheduler like the one we have presented.

The table shows that the <collect> task of the FILT benchmark has nearly 4000% variation on 40 cores (for 32 core run this variation is 495.30%). The <collect> task merges messages from multiple streams and forwards them to the subsequent component. Such a high variation indicates that at some point multiple messages were waiting to be merged, resulting in the sharp increase in latency seen in Fig. 9.

Benchmark	Task	Min	Max	Diff (%)
FFT	initP	1.1232s	1.9954s	77.65s
	stepP	10.3226s	15.2775s	48.00s
HIST	<collect>	0.9477s	1.5814s	66.86s
	<split>	0.8987s	4.2756s	375.74s
	split	3.9660s	5.0427s	27.15s
	calHist	22.3738s	28.7144s	28.34s
FILT	<collect>	0.6123s	27.0231s	4313.34s
	<filter>	0.1787s	0.4919s	175.19s
	<parallel>	0.4500s	1.5947s	254.38s
	<split>	0.3979s	11.5076s	2792.24s
	filt	134.2071s	470.1736s	250.33s
	split	1.0512s	6.3557s	504.62s

Table 1: Minimal and maximal task execution time on 40 cores

5 Related work

Verstraaten's SCC port of S-Net [32], where the core allocation is determined via static user annotations at the S-Net level, is closely related to our approach. In his approach the programmer must manually specify the allocation at compile time, which can be difficult and precludes system-wide load balancing under dynamic demand.

In our approach cores are allocated dynamically at the LPEL level beneath the S-Net runtime system. The approach involves keeping track of the system-wide workload and resource availability, enabling efficient task scheduling to maximise throughput and to reduce latency by dynamic load balancing. Also our approach can easily be extended towards dynamic power management.

The two approaches also differ in overhead. The distributed version of S-Net runs several extra tasks per core, such as the input manager, the output manager and the worker. This incurs extra overhead due to OS-level context switches. Our approach has only worker tasks, which considerably reduces the context switching overhead.

In [1] authors present a memory allocator called *scalloc* that is fast, multicore-scalable and provides low-fragmentation. The allocator is made-up of two parts; a frontend to manage memory in spans and a backend to manage empty spans. spans are same concept as superblocks in Hoard [3]. The spans are organised in 29 different size classes ranging from 16 bytes to 1MB. Any request for memory over 1MB is allocated directly from OS using mmap. Span-pool is a global concurrent data structure that holds spans in different pools using arrays and stack. Each span is used to fulfil memory request in terms of blocks, when all the blocks in span are freed, i.e. span has no allocated block it is returned to span-pool. With regard to memory allocation and deallocation we have similar approach, for example notion of ownership of memory block and separate lists to hold memory blocks that needs to be freed. For example, add block to local free list when allocation was done by same core, or add to remote free list otherwise, in case of *scalloc* it will be threads not cores. The main difference in our approach is that our allocator works across different instance of OS and uses less complex data structure and, can handle allocation bigger than 1MB in size.

In Intel's [9] *Privately Owned Public Shared Memory* (POP-SHM) approach, each core offers some private memory to share data with other cores. For computations, however, the data must be copied to private memory. In contrast, our middleware hides the details of memory management, enabling programming at a high level of abstraction.

Software Managed Cache-coherence (SMC) [33] provides coherent, shared, virtual memory, but it is the responsibility of the programmer to ensure that data is placed in the shared region and that operations to shared data are guarded by release/acquire calls. SMC is a library that provides coherent, shared memory, where as our middleware provides a high-level abstraction that simplifies programming.

MESH [23] is a framework for memory-efficient sharing. It uses remote method invocation to pass access to shared object between cores. The MESH framework uses POP-SHM for shared memory. It provides a higher level of abstraction than POP-SHM, but in contrast to our approach it does not provide a scheduler that is geared toward maximising throughput and reducing latency in streaming applications and relieving the programmer from worrying about load balancing.

Prell et al. [22] have presented an implementation of Go's [11] concurrency constructs on the SCC. Their approach uses Intel's RCCE [31] as communication library and employs work-stealing. The work shows that the implementation failed to scale due to limitation such as the number of simultaneously used channels and the size and number of data items exchanged over channels. In contrast, our middleware provides automatic load balancing and avoids these limitation. Furthermore, our middleware can easily be extended to exploit SCC-specific power management functionality.

6 Summary and Conclusion

We have presented a hierarchical memory management approach for tiled many-core processors. This memory management approach is capable to provide shared memory across multiple OS instances running on different cores. Based on that memory manager we were able to port the *Light-weight Parallel Execution Layer* (LPEL) to the Intel SCC research processor, making it the first execution middleware with dynamic load balancing to run on the SCC. We have studied the abstraction of communication, local cache deployment, and the resource-efficient use of the cores on the SCC research processor, which serves as an example of a tiled many-core architectures.

Our results show that our middleware is superior to an MPI-based implementation in throughput and latency. LPEL relies on multi-threading to offer high-performance lightweight tasks switching, which requires MPI to use TCP-based *sock* channels.

We also found that exploiting local caches is basically limited to non-shared data objects, resulting in inferior performance compared to the non-cached version. Using a cache-coherent tiled architecture may yield better performance. Further work is required to study the influence of cache locality and core interconnect topology for cache-coherent architectures.

References

1. M. Aigner, C. M. Kirsch, M. Lippautz, and A. Sokolova. Fast, multicore-scalable, low-fragmentation memory allocation through large virtual memory and global data structures. *arXiv preprint arXiv:1503.09006*, 2015.
2. Argonne National Laboratory. Mpich. `http://www.mcs.anl.gov/research/projects/mpi/` accessed 12-Dec-2013.
3. E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. *ACM Sigplan Notices*, 35(11):117–128, 2000.
4. S. Borkar. Thousand core chips: a technology perspective. In *DAC '07: Proc. 44th annual Design Automation Conference*, pages 746–749, New York, NY, USA, 2007. ACM.
5. S. Borkar and A. A. Chien. The future of microprocessors. *Commun. ACM*, 54:67–77, May 2011.
6. I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3), Aug. 2004.
7. G. Chrysos. Intel® xeon phi coprocessor-the architecture. *Intel Whitepaper*, 2014.
8. P. Ciancarini and T. Kielmann. Coordination models and languages for parallel programming. In *Proc. Int. Conf. on Parallel Computing (PARCO99)*, pages 3–17. Imperial College Press, 1999.
9. I. Corporation. The scckit 1.4.0 users guide. Technical report, Intel Corporation, March 2011. Revision 1.0.
10. T. Corporation. AMD FX Processor Product Brief, 2015. `http://www.tilera.com/files/drim__EZchip_LinleyDataCenterConference_Feb2015_7671.pdf` accessed 10-Mar-2015.
11. Golang.org. The go programming language. `http://golang.org` accessed 19-Apr-2014.
12. P. Greenhalgh. Big. little processing with arm cortex-a15 & cortex-a7. *ARM White paper*, 2011.
13. C. Grelck, S. Scholz, and A. Shafarenko. Asynchronous Stream Processing with S-Net. *Int. Journal of Parallel Programming*, 38(1):38–67, 2010.
14. C. Grelck, S.-B. Scholz, and A. Shafarenko. A Gentle Introduction to S-Net: Typed Stream Processing and Declarative Coordination of Asynchronous Components. *Parallel Processing Letters*, 18(2), 2008.
15. J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. V. D. Wijngaart, and T. Mattson. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 108–109, feb. 2010.
16. Intel Corporation. SCC External Architecture Specification (EAS). Technical report, Intel Labs, November 2010. Revision 1.1.

17. W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, Mar. 2004.
18. KALRAY Corporation. KALRAY MPPA MANYCORE Flyer, 2012. `http://www.kalrayinc.com/IMG/pdf/FLYER_MPPA_MANYCORE.pdf` accessed 10-01-2015.
19. B. W. Kernighan, D. M. Ritchie, and P. Ejeklint. *The C programming language*, volume 2. prentice-Hall Englewood Cliffs, 1988.
20. Microsoft. The Manycore Shift White Paper. Technical report, Microsoft Corporation, 2007.
21. V. Nguyen and R. Kirner. Demand-based scheduling priorities for performance optimisation of stream programs on parallel platforms. In J. Koodziej, B. Martino, D. Talia, and K. Xiong, editors, *Algorithms and Architectures for Parallel Processing*, volume 8285 of *Lecture Notes in Computer Science*, pages 357–369. Springer International Publishing, 2013.
22. A. Prell, T. Rauber, et al. Go's concurrency constructs on the scc. In *Proc. 6th Many-core Applications Research Community (MARC) Symposium*, pages 2–6, 2012.
23. T. Prescher, R. Rotta, and J. Nolte. Flexible sharing and replication mechanisms for hybrid memory architectures. In *Proc. 4th Many-core Applications Research Community (MARC) Symposium. Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam*, volume 55, pages 67–72, 2012.
24. D. Prokesch. A light-weight parallel execution layer for shared-memory stream processing. Master's thesis, Technische Universität Wien, Vienna, Austria, Feb. 2010.
25. C. Ramey. Tile-gx100 manycore processor: Acceleration interfaces and architecture. *Tilera Corporation*, 2011.
26. P. Salihundam, S. Jain, T. Jacob, S. Kumar, V. Erraguntla, Y. Hoskote, S. Vangal, G. Ruhl, and N. Borkar. A 2 Tb/s 6×4 Mesh Network for a Single-Chip Cloud Computer With DVFS in 45 nm CMOS. *Solid-State Circuits, IEEE Journal of*, 46(4):757–766, April 2011.
27. B. Schauer. Multicore Processors–A Necessity. *ProQuest Discovery Guides*, pages 1–14, 2008.
28. R. Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.
29. W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit: A language for streaming applications. In *Proc. 11th International Conference on Compiler Construction*, CC '02, pages 179–196, London, UK, UK, 2002. Springer-Verlag.
30. Tilera Corporation. Tile Processor Architecture Overview for the TILEPro Series, 2013. `http://www.tilera.com/scm/docs/UG120-Architecture-Overview-TILEPro.pdf` accessed 10-01-2015.
31. R. F. van der Wijngaart, T. G. Mattson, and W. Haas. Light-weight communications on Intel's single-chip cloud computer processor. *SIGOPS Oper. Syst. Rev.*, 45:73–83, February 2011.
32. M. Verstraaten. High-level Programming of the Single-chip Cloude Computer with S-Net. Master's thesis, University of Amsterdam, 2012.
33. X. Zhou, H. Chen, S. Luo, Y. Gao, S. Yan, W. Liu, B. Lewis, and B. Saha. A case for software managed coherence in manycore processors. In *Poster on 2nd USENIX Workshop on Hot Topics in Parallelism HotPar10*, 2010.

On Contracts and Sandboxes for JavaScript

Matthias Keil and Peter Thiemann

University of Freiburg, Freiburg, Germany
{keilr,thiemann}@informatik.uni-freiburg.de

Abstract. JavaScript is the language of the web. It is used by more than 89% of all the websites. Most of them rely on third-party libraries for connecting to social networks, feature extensions, or advertisement. Some of these libraries are packaged with the application, but others are loaded at run time from origins of different trustworthiness, sometimes depending on user input. Thus, managing untrusted JavaScript code has become one of the key challenges of present research on JavaScript.
This work is about *TreatJS* and the *TreatJS-Sandbox*.
TreatJS is a language embedded, higher-order contract system for JavaScript which enforces contracts by run-time monitoring. Beyond providing the standard abstractions for building higher-order contracts (base, function, and object contracts), *TreatJS*'s novel contributions are its guarantee of a non-interfering contract execution, its systematic approach to blame assignment, its support for contracts in the style of union and intersection types, and its notion of a parameterized contract scope, which is the building block for composable run-time generated contracts that generalize dependent function contracts.
The *TreatJS-Sandbox* is a language-embedded sandbox for full JavaScript. It enables scripts to run in a configurable degree of isolation with fine-grained access control. It provides a transactional scope in which effects are logged for review by the access control policy. After inspection of the log, effects can be committed to the application state or rolled back.

1 Introduction

We present the design and implementation of *TreatJS*, a language embedded, higher-order contract system for JavaScript which enforces contracts by run-time monitoring. *TreatJS* supports most features of existing systems and a range of novel features that have not been implemented in this combination before. No source code transformation or change in the JavaScript run-time system is required. In particular, *TreatJS* is the first contract system for JavaScript that supports the standard features of contemporary contract systems (embedded contract language, JavaScript in flat contracts, contracts as projections, full interposition using JavaScript proxies) in combination with the following three novel points.

1. *Noninterference.* Contracts are guaranteed not to exert side effects on a contract abiding program execution. A predicate is an arbitrary JavaScript

function, which can access the state of the application program but which cannot change it. An exception thrown by a predicate is not visible to the application program.

2. *Dynamic contract construction.* Contracts can be constructed and composed at run time using contract abstractions *without compromising noninterference.* A contract abstraction may contain arbitrary JavaScript code; it may read from global state and it may maintain encapsulated local state. The latter feature can be used to construct recursive contracts lazily or to remember values from the prestate of a function for checking the postcondition.
3. *New contract operators.* Beyond the standard contract constructors (flat, function, pairs), *TreatJS* supports object, intersection, and union contracts. Furthermore, contracts can be combined arbitrarily with the boolean connectives: conjunction, disjunction, and negation.

2 TreatJS by Example

TreatJS is implemented as a library so that all aspects of a contract can be specified using the full JavaScript language. The library relies on JavaScript proxies to guarantee full interposition for contracts. It further exploits JavaScript's reflective features to run contracts in a sandbox environment, which guarantees that the execution of contract code does not modify the application state. No source code transformation or change in the JavaScript run-time system is required.

In *TreatJS*, contracts are first-class values that can be stored or further composed. They are dormant until they are asserted to a value.

We start out with explaining *TreatJS*'s notation for base contracts and function contracts, and then move on to discuss intersection and union contracts. The implementation of the system is available on the Web. [1].

2.1 Base Contracts

The *base contract* is the fundamental building block for all other contracts. It is defined by a predicate, that is, a function returning a boolean value. In JavaScript, any function can be used as a predicate, because any return value can be converted to boolean. For example, the function *typeOfNumber* can serve as a predicate that checks whether its argument is a number.

```
function typeOfNumber (arg) {
  return (typeof arg) === ’number’;
};
```

To create a base contract from such a predicate, we apply the appropriate contract constructor to it.

```
var Num = Contract.Base (typeOfNumber);
```

[1] http://proglang.informatik.uni-freiburg.de/treatjs/

Here, ***Contract*** is the object that encapsulates the *TreatJS* implementation. Its ***assert*** method attaches a contract to a *subject*. Attaching a base contract applies the predicate to the value. If the result is true, ***assert*** returns the original value. Otherwise, ***assert*** signals a contract violation which blames the subject. The following example demonstrates both outcomes.

```
Contract.assert (1, Num); // accepted, returns 1
Contract.assert ('a', Num); // violation, blame subject 'a'
```

Figure 2.1 defines a number of base contracts for later use. Analogous to *Num*, the contracts *Bool* and *Str* check the type of their argument. Contract *Any* is a contract that accepts any value.

```
var Bool = Contract.Base (function (arg) {
  return (typeof arg) === 'boolean';
});
var Str = Contract.Base (function (arg) {
  return (typeof arg) === 'string';
});
var Any = Contract.Base (function (arg) {
  return true;
});
```

Fig. 1. Some utility contracts.

2.2 Function Contracts

While a base contract can specify finitary properties of a function f (like $f(1) = 0$), a *function contract* is needed to specify that a function uniformly maps numbers to booleans. A function contract is built from one or more contracts, zero or more for the arguments and one for the result of the function. Asserting a function contract to a non-function value immediately signals a contract violation. Asserting it to a function creates a wrapper function that asserts the argument contracts to the arguments of each call of the function and the result contract to the return value of each call.

As a running example, we consider the function *plus*, which applies the plus operator + to its arguments and returns the result.

```
function plus(x, y) {
  return (x + y);
}
```

The function contract *PlusNum* restricts a function's argument to a number and asserts that the result is a number.

```
var PlusNum = Contract.AFunction ([Num,Num], Num);
var plusNum = Contract.assert (plus, PlusNum);
```

In general, a JavaScript function has no fixed arity and arguments are passed to a function in a special array like object. Thus, a standard function contract takes two arguments. The first argument is an object contract that maps an argument to a contract. The second argument is a contract for the function’s return.

Contract.AFunction is the constructor for a simple function contract that takes an array of contracts for the arguments and a contract for the result of a function call as arguments.

The contracted function accepts any argument that satisfies the *Num* contract. If there is an argument that violates its contract, then the function contract raises an exception that blames the *context*, which is in this case the caller of the function that provides the wrong kind of argument. If the argument is ok, but the result contract fails, then blame is assigned to the *subject* (i.e., the function). Here are some examples that exercise *plusNum* as well as a broken version of it that returns a string.

```
plusNum (1, 2); // accepted, returns true
plusNum (’a’, ’b’); // violation, blame context ’a’

function plusBroken (x) {
  return (’’ + (x + y));
};
var plusNum2 = Contract.assert (plusBroken, PlusNum);
plusNum2 (1, 2); // violation, blame subject (function)
```

Higher-order contracts are also possible: the argument and result contracts may themselves be function contracts and so on, recursively. As an example, a function that takes a number and a numeric plus function as arguments and returns a number may be specified by the following contract.

```
var Add1Num =
  Contract.AFunction ([Num, PlusNum], Num);

function add1Broken (x, plus) {
  return plus(x, ’1’);
}
var add1BrokenNum = Contract.assert(add1Broken, Add1Num);
```

Higher-order contracts open up new ways for a function not to fulfill its contract. For example, the function *add1Broken* violates the contract *Add1Num*: the call *add1BrokenNum (1, plus)* signals a violation that blames the subject (the function) because it supplies the wrong kind of argument to its parameter *plus*.

Dually, a function that returns a function may be compromised. Consider the function *getAdd1* that fulfills the contract *GetAdd1*:

```
function getAdd1 (plus) {
  return function add1 (x) {
    return plus(x, 1);
  }
}
var GetAdd1 = Contract.AFunction ([PlusNum],
  Contract.AFunction ([Num], Num));
var add1Num = Contract.assert (getAdd1, GetAdd1) (plus);

add1Num (5); // accepted
add1Num ('a'); // violation, blame context 'a'
```

This example demonstrates that a function call that receives a suitable argument and returns a contract abiding result can still lead to a contract violation if the result is misused.

2.3 Intersection and Union Contracts

In the previous section, the function *plus* was contracted with *PlusNum* to restrict the arguments to numbers. Indeed, *plus* fulfills this contract so that we might say it has type $\mathit{Num}, \mathit{Num} \to \mathit{Num}$. However, the plus operator of JavaScript is overloaded and does not restrict its arguments to numbers: it works just as well if one argument is a string. Thus, *plus* also has type $\mathit{Str}, \mathit{Str} \to \mathit{Str}$.

TreatJS provides a corresponding constructor for intersection contracts.

```
var PlusStr = Contract.AFunction ([Str,Str], Str);
var PlusNumStr = Contract.Intersection (PlusNum, PlusStr);
var plusNumStr = Contract.assert (plus, PlusNumStr);
```

The function *plusNumStr* may be applied to number or string values and promises to return a either a number or a string, depending on its arguments. The context is blamed if it provides the function with an argument that does not fulfill the expectations. The subject is blamed if the function does not fulfill both constituent contracts.

Generally, the subject f of an intersection contract $C \cap D$ must fulfill both contracts C and D. If $C = C_1 \to C_2$ and $D = D_1 \to D_2$ are both function contracts, then any argument to f has to fulfill $C_1 \cup D_1$. Additionally, the context must be prepared to handle a value satisfying $C_2 \cup D_2$. In case the argument contracts overlap (i.e., $C_1 \cap D_1 \neq \emptyset$), then applying the function to an element in their intersection must yield a result that satisfies both, C_2 and D_2. As an example for the case where $C_1 \cap D_1 \neq \emptyset$, consider

```
var StrAny = Contract.AFunction ([Str,Any], Str)
var AnyStr = Contract.AFunction ([Any,Str], Str)
var PlusAny = Contract.Intersection (StrAny, AnyStr);
```

which is another valid typing for the *plus* function.

Just like intersections, union contracts are also applicable to functions. They also mimick union types as closely as possible. That is, a function satisfies a union of two function contracts if it satsifies either of them.

```
var TestPlus = Contract.Union(
  Contract.AFunction([PlusNum], Num),
  Contract.AFunction([PlusStr], Str));
```

A function which satisfied such a contract is either a function that accepts a plus function which satisfies *PlusNum* and returns a number or by one that accepts a plus function that satisfies *PlusStr* and returns a string value. As an example consider the *testPlus* function.

```
function testPlus (plus) {
  return plus(1, 2);
}
var testPlusNumStr = Contract.assert(testPlus, TestPlus);
```

Because the context do not know which kind or arguments *testPlus* supplies to its *plus* argument, he has to call *testPlus* with a plus function that satisfiers the intersection between *PlusNum* and *PlusStr*.

2.4 Dependent Contracts

A dependent contract is a contract on functions where the range portion depends on the function argument. The contract for the function's range can be created with a contract abstraction, a contract that returns a contract. This abstration is invoked with the caller's argument. so that the returned contract may refer to those values.

TreatJS's dependent contract operation only builds a range contract in this way; it does not check the domain as checking the domain may be achieved with a conjunction with another function contract.

For example, a dependent contract may be used to specify that the arguments type of function *add1* corresponds to the type of the functions return.

```
var SameType = Contract.SDependent(function(input) {
  return Contract.Base(output) {
    return (typeof input) === (typeof output).
  }
});
```

The contract receives the input arguments and returns a contract for the range that checks that the type of the input is identical to the type of the result. When calling a function contracted with the dependent contract *SameType*, the abstraction is invoked on the arguments and the resulting contract is imposed on the return value.

3 Sandboxing of Predicates

TreatJS is implemented as a library so that all aspects of a contract can be specified using the full JavaScript language. For example, the base contract *Num* checks its argument to be a number.

```
var Num = Contract.Base(function (arg) { {
  return (typeof arg) === 'number';
});
```

Asserting a base contracts to a value causes the predicate to be checked by applying the predicate to the value.

```
Contract.assert(1, Num); // accepted
```

However, predicates are attempted not to influence the program state in any way. A monitored program execution should either throw a contract violation or evaluate to the same result as without contracts.

TreatJS relies on the sandbox presented in this work to guarantee that the execution of contract code does not interfere with the contract abiding execution of the host program.

To illustrate, we use a modified *Num* contract.

```
var NumBroken = Contract.Base(function(arg) {
  type = (typeof arg);
  return type === 'number';
});
```

When asserting *NumBroken*, sandboxing intercepts the unintended write to the global variable *type* in the following code and throws an exception.

As read-only access to objects and functions is safe and useful in many contracts, TreatJS facilitates making external references visible inside of the sandbox. For example, the *Ary* contract below references the global object *Array*.

```
var Ary = Contract.With(
  {Array:Array},
  Contract.Base(function (arg) {
    return (arg instanceof Array);
}));
```

4 Transaction-based Sandboxing: A Primer

Today's state of the art in securing JavaScript application that include code from different origins is an all-or-nothing choice. Browsers apply protection mechanisms, such as the same-origin policy or the signed script policy, so that scripts either run in isolation or gain full access.

While script isolation guarantees noninterference with the function of the application as well as preservation of data integrity and confidentiality, there are scripts that must have access to part of the application state to function meaningfully. As all included scripts run with the same authority, the application script cannot exert fine-grained control over the use of data by an included script.

Transactional sandboxing is inspired by the idea of transaction processing in database systems and transactional memory. Each sandbox implements a transactional scope the content of which can be examined, committed, or rolled back. Its design is inspired by revocable references and SpiderMonkey's compartment concept. Our sandbox provides the following novel features:

1. *Language embedded.* The sandbox is implemented as a library in JavaScript. It handles the full JavaScript language (ES5) including its dynamic features. No source code transformation or change in the JavaScript run-time system is required.
2. *Full interposition.* Our sandbox adapts SpiderMonkey's compartment concept[2] and runs code in isolation to the application.
3. *Transaction-based sandboxing.* The sandbox provides a transactional scope. A proxy-based membrane makes objects accessible inside the sandbox, performs effect logging, and enables locally visible modifications. After inspection of the log, effects can be committed to the application state or rolled back.

The implementation of the system is available on the Web[3].

4.1 Cross-Sandbox Access

We consider operations on binary trees as defined by *Node* in Figure 2 along with some auxiliary functions. As an example, we perform operations on a tree consisting of one node and two leaves. All value fields are initially *0*.

```
var root = new Node(0, new Node(0), new Node(0));
```

Next, we create a new empty sandbox by calling the constructor *Sandbox*. Its first parameter acts as the global object of the sandbox environment. It is wrapped in a proxy to mediate all accesses and it is placed on top of the scope chain for code executing inside the sandbox. The seconds parameter is a configuration object. A sandbox is a first class value that can be used for several executions.

```
var sbx = new Sandbox(this, {/* some parameters */});
```

One use of a sandbox is to wrap invocations of function objects. To this end, the sandbox API provides methods *call*, *apply*, and *bind* analogous to methods from ***Function***.*prototype*. For example, we may call *setValue* on *root* inside of *sbx*.

[2] SpiderMonkey creates one heap for each website, initially introduced to optimize garbage collection. All objects created by a website are only allowed to touch objects in the same compartment. Proxies are used as cross compartment wrappers to make objects accessible in other compartments.

[3] `https://github.com/keil/Sandbox`

```
1  function Node (value, left, right) {
2    this.value = value;
3    this.left = left;
4    this.right = right;
5  }
6  Node.prototype.toString = function () {
7    return (this.left?this.left + ”, ”:””) + this.value +(this.right?”, ”+this.right:””);
8  }
9  function heightOf (node) {
10   return Math.max(((node.left)?heightOf(node.left)+1:0), ((node.right)?heightOf(
         node.right)+1:0));
11 }
12 function setValue (node) {
13   if (node) {
14     node.value=heightOf(node);
15     setValue(node.left);
16     setValue(node.right);
17   }
18 }
```

Fig. 2. Implementation of *Node*. Each node object consists of a value field, a left node, and a right node. Its prototype provides a *toString* method that returns a string representation. Function *heightOf* computes the height of a node and function *setValue* replaces the value field of a node by its height, recursively.

```
21 sbx.call(setValue, this, root);
```

The first argument of *call* is a function object that is decompiled and redefined inside the sandbox. This step erases the function's free variable bindings and builds a new closure relative to the sandbox's global object. The second argument, the receiver object of the call, as well as the actual arguments of the call are wrapped in proxies to make these objects accessible inside of the sandbox.

The wrapper proxies mediate access to their target objects outside the sandbox. Reads are forwarded to the target unless there are local modifications. The return values are wrapped in proxies, again. Writes produce a *shadow value* that represents the sandbox-internal modification of an object. Initially, this modification is only visible to reads inside the sandbox.

Native objects, like the *Math* object in line 10, are also wrapped in a proxy, but their methods cannot be decompiled because there exists no string representation. Thus, native methods must either be trusted or forbidden. Fortunately, most native methods to not have side effects, so we chose to trust them.

Given all the wrapping and sandboxing, the call in line 21 did not modify the *root* object:

```
22 root.toString(); // returns 0, 0, 0
```

But calling *toString* inside the sandbox shows the effect.

```
sbx.call(root.toString, root); // return 0, 1, 0
```

4.2 Effect Monitoring

During execution, each sandbox records the effects on objects that cross the sandbox membrane. The resulting lists of *effect objects* are accessible through *sbx .effects*, *sbx.readeffects*, and *sbx.writeeffects* which contain all effects, read effects, and write effects, respectively. All three lists offer query methods to select the effects of a particular object.

```
sbx.call(heightOf, this, root);
var rects = sbx.effectsOf(this);
print(”;;; Effects of this”);
rects.foreach(function(i, e) {print(e)});
```

The code snippet above prints a list of all effects performed on *this*, the global object, by executing the *heightOf* function on *root*. The output shows the resulting accesses to *heightOf* and *Math*.

```
;;; Effects of this
(1425301383541) has [name=heightOf]
(1425301383541) get [name=heightOf]
(1425301383543) has [name=Math]
(1425301383543) get [name=Math]
...
```

The first column shows a timestamp, the second shows the name of the effect, and the last column shows the name of the requested parameter. The list does not contain write accesses to *this*. But there are write effects to *value* from the previous invocation of *setValue*.

```
var wectso = sbx.writeeffectsOf(root);
print(”;;; Write Effects of root”);
wectso.foreach(function(i, e) {print(e)});

;;; Write Effects of root
(1425301634992) set [name=value]
```

4.3 Inspecting a Sandbox

The state inside and outside of a sandbox may diverge for different reasons. We distinguish changes, differences, and conflicts.

A *change* indicates if the sandbox-internal value has been changed with respect to the outside value. A *difference* indicates if the outside value has been modified after the sandbox has concluded. For example, a difference to the previous execution of *setValue* arises if we replace the left leaf element by a new subtree of height 1 outside of the sandbox.

```
root.left = new Node(new Node(0), new Node(0));
```

Changes and differences can be examined using an API that is very similar to the effect API. There are flags to check whether a sandbox has changes or differences as well as iterators over them.

A *conflict* arises in the comparison between different sandboxes. Two sandbox environments are in conflict if at least one sandbox modifies a value that is accessed by the other sandbox later on. We consider only Read-After-Write and Write-After-Write conflicts.

To demonstrate conflicts, we define a function *appendRight*, which adds a new subtree on the right.

```
function appendRight (node) {
  node.right = Node('a', Node('b'), Node('c'));
}
```

To recapitulate, the global *root* is still unmodified and prints *0,0,0,0,0*, whereas the *root* in *sbx* prints *0,0,0,1,0*. Now, let's execute *appendRight* in a new sandbox *sbx2*.

```
var sbx2 = new Sandbox(this, {/* some parameters */});
sbx2.call(appendRight, this, root);
```

Calling *toString* in *sbx2* prints *0,0,0,0,b,a,c*. However, the sandboxes are *not* in conflict, as the following command show.

```
sbx.inConflictWith(sbx2); // returns false
```

While both sandboxes manipulate *root*, they manipulate different fields. *sbx* recalculates the field *value*, whereas *sbx2* replaces the field *right*. Neither reads data that has previously been written by the other sandbox. However, this situation changes if we call *setValue* again, which also modifies *right*.

```
sbx.call(setValue, this, root);
var cofts = sbx.conflictsWith(sbx2); // returns a list of conflicts
cofts.foreach(function(i, e) {print(e)});
```

It documents a read-after-write conflict:

```
Confict: (1425303937853) get [name=right]@SBX001 - (1425303937855) set [
    name=right]@SBX002
```

4.4 Transaction Processing

The *commit* operation applies select effects from a sandbox to its target. Effects may be committed one at a time by calling *commit* on each effect object or all at once by calling *commit* on the sandbox object.

```
sbx.commit();
root.toString(); // returns 0, 1, 0, 2, 0
```

The *rollback* operation undoes an existing manipulation and returns to its previous configuration before the effect. Again, rollbacks can be done on a per-effect basis or for the sandbox as a whole. However, a rollback did not remove the shadow object. Thus, after rolling back, the values are still shadow values in *sbx*.

```
51 sbx.rollabck();
52 root.toString(); // returns 0, 1, 0, 2, 0
53 sbx.call(toString, this, root); // returns 0, 0, 0, 0, 0
```

The *revert* operation resets the shadow object of a wrapped value. The following code snippet reverts the *root* object in *sbx*.

```
54 sbx.revertOf(root);
```

Now, *root*'s shadow object is removed and the origin is visible again in the sandbox. Calling *toString* inside of *sbx* returns *0,1,0,2,0*.

4.5 Transparent Sandboxing

Transparent sandboxing is a special mode of our sandbox. It deactivates the shadowing of write operations so that modifications apply directly to the target objects. As those modifications are performed inside the sandbox, write effects are still logged, so that they can be inspected and rolled back as usual. It can be enabled by changing the *transparent* flag in the sandbox configuration. Here is an example:

```
55 var tsbx = new Sandbox(this, {transparent:true});
56 tsbx.call(setValue, this, root);
```

Calling *toString* demonstrates the difference to the standard, non-transparent sandbox: All changes of line 56 are visible.

```
57 root.toString(); // returns 0, 1, 0, 2, 0
```

Calling *tsbx.rollback();* resets all modifications of *tsbx*. Afterwards, *root* prints *0,0,0,2,0*.

4.6 Pre-state Snapshot

The *snapshot* mode instructs the membrane to clone target objects at initialization time and to use the clone as shadow object. The snapshot enables to *rebase* the sandbox to its initialization state.

A snapshot can be triggered by including the object in the third argument of the sandbox constructor, the snapshot array.

```
58 var ssbx = new Sandbox(this, {/* some parameters */}, [root]);
```

The sandbox can be used as before.

```
59 ssbx.call(setValue, this, root);
60 ssbx.call(root.toString, root); // returns 0, 1, 0, 2, 0
```

Remember, the original *root* object prints *0,0,0,2,0*. Now, let's do some changes, for example by calling *setValue(root);*.

Both representations prints *0,1,0,2,0*. But if one would rebase the sandbox to its initial state, by calling *ssbx.rebase();*, the values go back to the version that exist at initialization time.

```
61 ssbx.call(root.toString, root); // returns 0, 0, 0, 2, 0
```

4.7 Wrapping

The methods *call* and *apply* are shortcuts. Internally, they call a *wrap* method to redefine the function inside of the sandbox and apply the corresponding method from ***Function***.*prototype* to it. The following example shows an alternative to the call in line 21.

```
62 sbx.wrap(setValue).call(this, root);
```

But *wrap* can also be used independently. One example is to obtained a sandboxed version of *root*.

```
63 var sbxroot = sbx.wrap(root);
```

The returned object is wrapped in the sandbox membrane and identical to the object visible inside of the sandbox. Each read access on *sbxroot* returns another sandbox object and each each write access causes an effect. All sandbox features like commit, rollback, and effect logging remain active.

Calling *toString* on *sbxroot* returns *0,1,0,2,0*. The method call illustrates that *sbxroot* is the modified object that occurs in *sbx*. Nevertheless, *sbxroot* can be used like any other object.

This feature allows us to extend an existing data structure with transactional features. For example, instead of defining *root* directly, a developer could define it as follow.

```
64 var sbx3 = new Sandbox(this, {/* some parameters */});
65 var root = sbx3.wrap(new Node(0, new Node(0), new Node(0)));
```

Proxies guarantee that the new *root* object performs as usual, for example when calling *setValue(root)*. But it enables to use all sandbox features in addition, e.g. to commit changes or to roll back.

5 Related Work

Contract Monitoring TreatJS [7] is a language embedded, dynamic, higher-order contract system implemented in JavaScript. Its development is based on a novel denotational semantics of contracts and on a blame calculus [6] that enables higher-order contract with unrestricted intersection and union of contracts. The specification for intersection and union contracts is strongly inspired by their type-theoretic counterparts. This connection tightly integrates statically and dynamically typed worlds which may be beneficial for future integration in a gradual type system.

Effect Monitoring JSConTest [2] is a framework that helps to investigate the effects of unfamiliar JavaScript code by monitoring the execution and by summarizing the observed access traces to access permission contracts. It comes with an algorithm [3] that infers a concise effect description from a set of access paths and it enables the programmer to specify the effects of a function using access permission contracts.

JSConTest2 [5] is a redesign and a reimplementation of JSConTest using JavaScript proxies. The new implementation addresses shortcomings of the previous version. In particular, the proxy-based implementation guarantees full interposition for the full language and for all code regardless of its origin, including dynamically loaded code and code injected via *eval*.

JavaScript Proxies Object equality becomes an issue for non-interference when the executed code ends up in a mixture between wrapper and target. The problem arises if an equality test between wrapper and target returns false instead of true. The work of Keil et al. [4] examines this problem and presents a modification of the underlying VM with respect to object equality and introduces new transparent proxies that fit better to this use case.

6 Conclusion

We presented *TreatJS*, a language embedded, dynamic, higher-order contract system for full JavaScript. *TreatJS* extends the standard abstractions for higher-order contracts with intersection and union contracts, boolean combinations of contracts, and parameterized contracts, which are the building blocks for contracts that depend on run-time values. *TreatJS* implements proxy-based sandboxing for all code fragments in contracts to guarantee that contract evaluation does not interfere with normal program execution. The only serious impediment to full noninterference lies in JavaScript's treatment of proxy equality, which considers a proxy as an individual object.

The *TreatJS-Sandbox* runs JavaScript code in a configurable degree of isolation with fine-grained access control. It provides a transactional scope in which effects are logged for inspection. Effects can be committed to the application state or rolled back.

Both systems are implemented as a JavaScript library. No source code transformation or adaption in the JavaScript run-time system is required. All aspects are accessible through a sandbox API.

References

1. John Boyland, editor. *ECOOP 2015 - Object-Oriented Programming - 29th European Conference*, volume ?, Prague, Czech Repulic, July 2015. LIPICS.
2. Phillip Heidegger, Annette Bieniusa, and Peter Thiemann. Access permission contracts for scripting languages. In John Field and Michael Hicks, editors, *Proc. 39th ACM Symp. POPL*, pages 111–122, Philadelphia, USA, January 2012. ACM Press.
3. Phillip Heidegger and Peter Thiemann. A heuristic approach for computing effects. In Judith Bishop and Antonio Vallecillo, editors, *Proc. 49th TOOLS*, volume 6705 of *LNCS*, pages 147–162, Zurich, Switzerland, June 2011. Springer.
4. Matthias Keil, Sankha Narayan Guria, Andreas Schlegel, Manuel Geffken, and Peter Thiemann. Transparent object proxies in javascript. In Boyland [1], pages 149–173.
5. Matthias Keil and Peter Thiemann. Efficient dynamic access analysis using JavaScript proxies. In *Proceedings of the 9th Symposium on Dynamic Languages*, DLS '13, pages 49–60, New York, NY, USA, 2013. ACM.
6. Matthias Keil and Peter Thiemann. Blame assignment for higher-order contracts with intersection and union. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 375–386, New York, NY, USA, 2015. ACM.
7. Matthias Keil and Peter Thiemann. Treatjs: Higher-order contracts for javascripts. In Boyland [1], pages 28–51.

On the Specification of Real-time Properties of Streaming Networks [*]

Raimund Kirner, Simon Maurer

University of Hertfordshire
r.kirner, s.maurer@herts.ac.uk

Abstract. Cyper-physical systems (CPS) are networked embedded systems, having often real-time requirements for individual control tasks. The complexity of CPS due to concurrency can be reduced by modelling it as a streaming network, providing an implicit local synchronisation mechanism.

In this paper we show that specifying real-time requirements for such streaming networks is not straight forward. Especially specifying latency is challenging due to their global context from which they arise. Analysing models at requirements and system design level, we provide practical solutions to the specification of the timing behaviour of such models of streaming networks.

1 Introduction

Streaming networks consists of processing nodes connected via communication channels. Since this communication channels have a single reader and a single writer, streaming networks are a well-recognised for their benefit of coping with the complexity of concurrent systems.

This strength of streaming networks makes them also an interesting paradigm to apply to cyber-physical systems (CPS). CPS are networked embedded systems, thus exposing naturally a high level of concurrency [9]. At the same time CPS have often real-time requirements, often for local subsystems, but sometimes also at a global level.

Lee has proposed coordination languages as a means to cope with the complexity of CPS [9]. We support this observation and also propose to use the streaming network paradigm such the underlying concept for such a coordination language.

In this paper we do not focus on the overall design of a coordination language well suited for CPS. But rather, we focus on the underlying streaming network paradigm and discuss how it suits the specification of real-time requirements. As we show in this paper, it is not so easy to specify real-time properties for streaming networks in a simple and resource-efficient way.

In Section 2 we study the specification of real-time properties for real-time systems of simple structure, exposing the difference between modelling timing behaviour at requirements level and at system design level. In Section 3 we show the challenges that

[*] This work was partially supported by COST Action IC1202: "Timing Analysis On Code-Level" (TACLe). The research leading to these results has received funding from the FP7 ARTEMIS-JU research project "ConstRaint and Application driven Framework for Tailoring Embedded Real-time Systems" (CRAFTERS).

arise when aiming to annotate real-time properties in streaming networks and propose concrete solutions. Section 4 shows some examples of streaming networks applied to CPS. A selection of related work is discussed in Section 5. Section 6 concludes the paper.

2 Development of Real-time Systems

In order to expose the challenge of specifying real-time properties of streaming networks we first review some fundamentals of developing real-time programs. There are basically two extreme contexts for modelling extra-functional behaviour like real-time requirements:

1. a high-level model that describes how the system is expected to fit into the environment. We call such a model a requirements specification or *requirements model*.
2. a low-level model that describes how the system is designed, with various levels of detail. We call such a model a *design model*.

Depending on the concrete software development processes of specific application domains, there are more fine-grained distinctions of modelling levels. However, for the sake of simplicity, we focus only on these two fundamental ones. Furthermore, in the following we discuss the timing-related aspects of such system models.

To understand modelling of real-time systems from its fundamentals, we assume a simple system structure where a particular services is expected to work on input, derived from sensors, to produce an output for an actuator. This fundamental structure is shown in the top of Figure 1.

Focusing on the system requirements we derive a requirements model. In the context of real-time systems such a requirements model has to include the specification of extra-functional properties, especially timing requirements. The modelling aspects of timing requirements are shown in the bottom half of Figure 1. Based on the simple system structure we can express timing requirements as a tuple $\langle I_x, S_y, O_z, Treq_{x,y,z} \rangle$, meaning that each timing requirement $Treq_{x,y,z}$ spans from a particular input I_x to an output O_z, characterising a service S_y. In the timing domain these requirements typically include the processing rate and the latency of a service. The processing rate is sometimes also called throughput. The variation in rate or latency is called jitter, and can also be included in the timing requirements. While often jitter is exclusively associated with latency, we consider it also applicable to processing rate, especially for such real-time systems where the processing rate is more important than the latency, e.g., in video streaming applications without control loops.

To give an example of a timing requirement, we consider the maximum latency for a service, also called a *deadline*. Deadlines are called firm if the utility of a service abruptly drops after exceeding the deadline, otherwise it is called a soft deadline [8]. Figure 2 shows the specification of a deadline for a service S_y from input I_x to output O_z. It is important to include the data-flow path attached to the timing requirement, since the same service, for example, might also write to another output with a different deadline attached to it. The deadline shown in Figure 2 is relative, i.e., the maximum

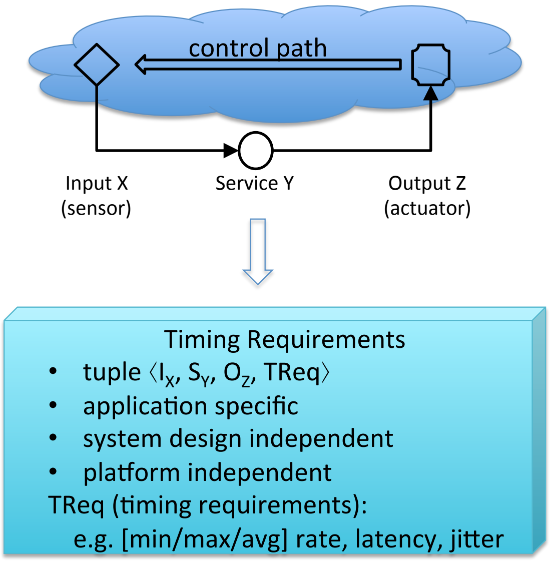

control path
Input X
(sensor)
Service Y
Output Z
(actuator)
Timing Requirements
• tuple ⟨I$_X$, S$_Y$, O$_Z$, TReq⟩
• application specific
• system design independent
• platform independent
TReq (timing requirements):
e.g. [min/max/avg] rate, latency, jitter

Fig. 1. Derivation of real-time requirements from application context

latency is measured against the trigger instant at input I_x. Each trigger instant of input I_x results into a different absolute deadline:

$$t_{deadline} = t_{input} + t_{deadline,rel}$$

Figure 3 shows the transition from the requirements model to the design model. While the requirements model is solely focused on the demands imposed by the application environment of the real-time control system, the design model shifts its focus to the details of how to build the system, resulting in a design-specific model. As shown in Figure 3, the design model can be expressed at different abstraction stages, ranging from an implementation-independent model to an implementation-dependent model. The implementation-dependent design model includes imposed properties like the choice of platform. But the implementation-dependent design model might also be enriched by annotations, derived from behaviour analysis, making behavioural properties, resulting from the implementation choices, explicit. Figure 4 shows a timing requirement of latency (response time) attached to the design model.

The fundamental difference between Figure 4 and Figure 2 is that the abstract specification of a service in Figure 2 has been replaced by a concrete processing node realising that service. The relation between services and processing nodes realising them is, in general, $n : m$ and not necessarily $1 : m$ or $n : 1$.

Here we want to stress that ideally a design model already reflects the real-time behaviour of the system at a platform-independent stage, if possible even at an implementation-independent stage. The benefit of a platform-independent design model is that the system behaviour can be reasoned about independently of the platform choice, making a

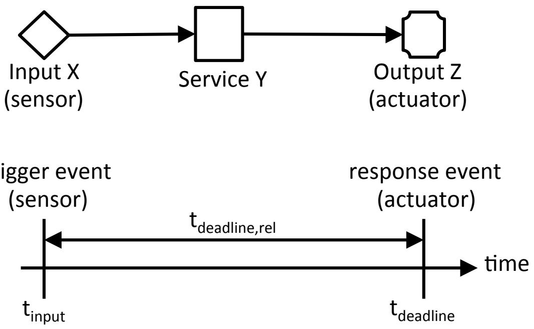


Fig. 2. Example of timing requirements: deadline (maximum latency)

correctness verification more robust against changes of the platform at a later development stage.

However, including the specification of the timing behaviour into the platform-independent design model also comes with a cost. We have to ensure, after the platform choice, that the specified timing behaviour is actually implementable with the chosen platform. While this is nothing surprising by itself, there is a fundamental difference of whether we have to fulfil the timing specification of the requirements model or whether we have to fulfil the timing requirements of an platform-independent design model. The latter may impose additional design-specific constraints that can rise resource constraints which are not imposed by the application context itself. In Section 3 this aspect is discussed in more detail within the context of timed streaming networks.

3 Specification of Complex Systems

In Section 2 we have discussed the specification of timing requirements of real-time systems with a very simple structure. In the context of networked cyber-physical systems we have to deal with much more complex application structures. In the following we discuss challenges of modelling extra-functional properties for both, requirements and design models.

3.1 Timed Requirements Models

One of the challenges of requirements modelling is that requirements have to be developed in a modular way in order to cope with complexity. Like the example shown in Figure 5, services of a system tend to be described in a cascaded way. We cannot use the simple concept of Section 2 where a service is used to characterise the information processing between the inputs and outputs of a system.

Instead of linking services directly to sensor inputs and actuator outputs, we have to use some form of system interfaces. Regardless of the concrete specification methods being used in practice, we abstract from them by using the generic concept of ports. Sensors, actuators, and individual services can now be characterised locally within the

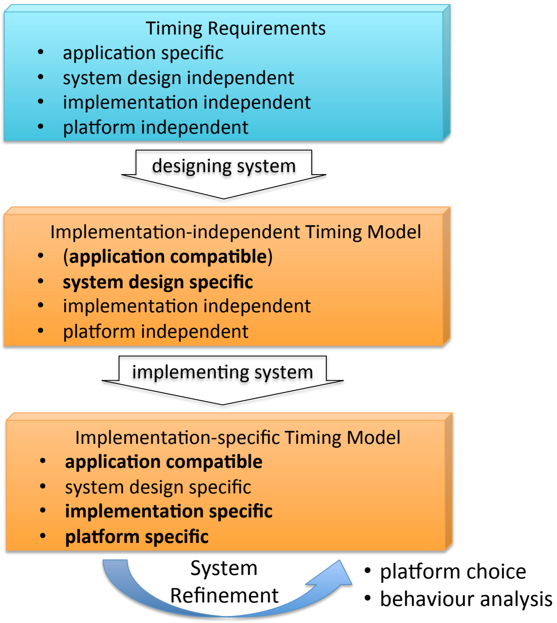


Fig. 3. Derivation of real-time model from application context and refined by analysis

perimeter of their interface ports p_i. Figure 5 shows the use of ports. Not only can the information input by a sensor now be discussed independently of its use, the figure also shows the characterisation of two services SV_1 and SV_2, which are cascaded, i.e., one input of SV_2 is not connected to a sensor but rather to the output of another service. SV_2 might rely on SV_1 on a rather weak basis to provide a refined quality of service, but it can be also the case that SV_2 strictly relies on SV_1 to provide any useful service at all.

The challenge of modelling timing requirements of cascaded services is that timing requirements in their purest form are imposed by the environment and are independent of any internal structuring of the computer system. For example, in Figure 5 there might be a certain relative deadline $d_{2,1}$ from sensor S_2 to actuator A_1. At the same time, there might be a relative different deadline $d_{1,1}$ from sensor S_1 to actuator A_1. The problem with the cascaded services S_1 and S_2 is that one cannot naturally assign them fractions of $d_{1,1}$ and $d_{2,1}$ without creating artificial constraints on the flexibility of the use of resources.

To characterise the latency requirements in their purest form one would have to use a kind of path-based characterisation of latency requirements. Instead of the approach in Section 2 where a timing requirement $Treq_{x,y,z}$ was associated with a context tuple $\langle I_x, S_y, O_z \rangle$, we would now have to match a timing requirement $Treq_p$ with a path specifier. A path specifier pth is a sequence $pth = (a_1, a_2, a_3, \ldots a_n)$ where each element a_i of the sequence is either a service, a port, or an IO node: $a_i \in SERVICES \cup PORTS \cup IO$. A path specifier may also only incompletely describe a path, by which

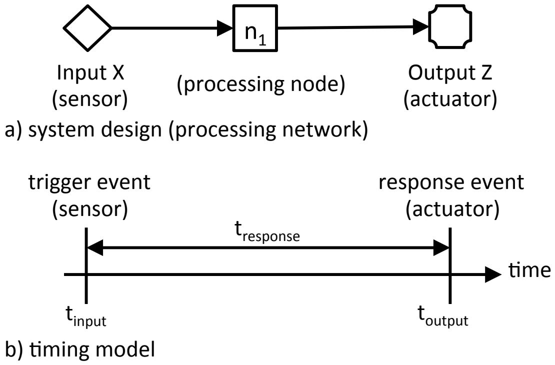


Fig. 4. Timing model derived from application context: Response time

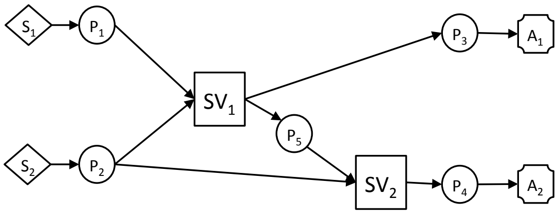


Fig. 5. Specification of multiple Services (Requirements Model)

multiple paths would match the specification. This can be used to assign a requirement to a group of paths. For example, the path specifier (S_2, P_2, P_4, A_2) matches with two concrete paths of Figure 5: $(S_2, P_2, SV_1, P5, SV_2, P_4, A_2)$ and $(S_2, P_2, SV_2, P_4, A_2)$.

The good news is that modelling requirements of processing rates can be done locally and propagated over the network. This approach, for example, is used in Simulink from Mathworks [10]. Nevertheless, clear semantic rules are needed to specify what happens at the interface between different processing rates [11].

Summarising, the challenge of specifying timing requirements for streaming networks is to express them in a pure form, i.e., only implied by the application environment and not by any internal system structuring decisions.

3.2 Timed Design Models

In this section we discuss the issues of specifying timing requirements for design models. As discussed before, the design model focuses on the behaviour of the realised system, where the realised system consists of inter-linked processing nodes with in the general case an $n : m$ mapping between services and processing nodes.

The challenge of how to describe latency in case of cascaded processing nodes is related with the challenge of specifying latency for cascaded services mentioned in Section 3.1. The additional challenge for the design model is that it tends to be more complex than the requirements model in case of individual services being realised by multiple processing nodes. Figure 6 shows a streaming network with processing nodes fed by multiple sensors and contributing to the control of multiple actuators.

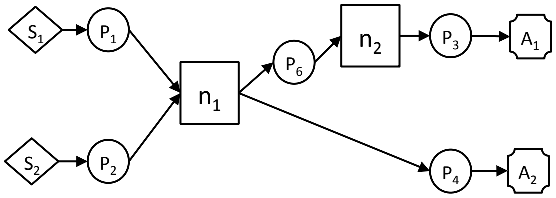


Fig. 6. Processing network with multiple sensors and actuators (Design Model)

What we ultimately want to obtain is a design model which has a clear timing semantics, i.e., it is known what the timing behaviour of different components will be, at least at a certain course granularity level. This is an aim somehow similar to the idea of the *Precision Timed Machine* (PRET) whose machine code has a timed semantics [7, 6]. However, there is a slight difference. With PRET the focus is on a well-specified processor platform with built-in timing semantics. The machine code for PRET would be a platform-specific design model of the computer system.

However, with our ambition for design models of streaming networks we would like to have a platform-independent design model with well-specified temporal behaviour. This means, we would like to know how the system is going to behave, regardless of the implementation details and even more, regardless of the chosen hardware platform. With streaming networks we have the challenge that at a particular position of the network messages originating from different sources can pass through, thus we generally cannot assign the processing latency of a path to any particular place in the network. So if we want to specify the timing behaviour directly at the design model rather than having a separate list of timing constraints, we would have to split the overall latency into multiple local latency values, assigned to individual sections of the streaming network.

For the split latency values there are two different semantics possible:

Summative latencies: the absolute latency along a path from the input to the output is the sum of all the local latency values along that path. Summative latencies do not require a platform-specific design model, so they can be also specified for a platform-independent design model.

Local absolute latencies: each local latency value describes the absolute local latency along a certain subsection of a processing path from the system input to the output. Absolute local latencies require a platform-specific design model, so they cannot be specified for a platform-independent design model.

To be most descriptive, the local latencies have be both, summative latencies as well as local absolute latencies.

To give an example, we assume that the streaming network shown in Figure 6 has as requirement the following absolute latencies from sensor input to actuator output:

	A_1	A_2
S_1	10 ms	4 ms
S_2	10 ms	4 ms

Having only a platform-independent design model, we can still decompose these end-to-end latencies into summative latencies and map them to the streaming network of Figure 6. Figure 7 shows summative latencies for the given example mapped to the streaming network. In this small example we have been able to derive the summative latencies manually. For larger graphs a systematic mapping method would be necessary to use.

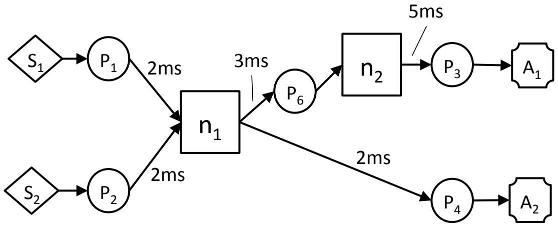


Fig. 7. Processing network with timed semantics (Design Model)

Using such summative latencies for a platform-independent design model are adequate to specify the end-to-end latencies by means of local annotation. However, it would be problematic to interpret them as absolute latencies for local sections of the streaming network. By doing so we would impose additional synthetic resource constraints for implementation and platform choice, not justified by the requirements. Thus our proposal is that for the platform-independent design model we interpret the local latency specifications in general only as summative latency specifications.

From Summative Latencies to Local Absolute Latencies As soon as we have derived a platform-specific design model, we are able to use performance or worst-case execution time (WCET) analysis to refine the model and replace the original summative latencies by another set of latencies that at the summative level are equivalent to the summative latencies of the platform-independent design model, but now also have a local absolute latency semantics.

Using such a refinement step towards the platform-specific design model avoids the introduction of synthetic resource constraints while still providing a fundamental timing semantics at the platform-independent design model. This approach is summarised in Figure 8.

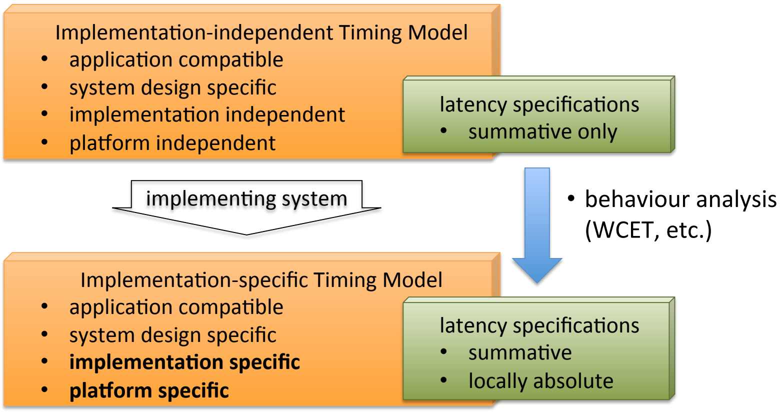


Fig. 8. Derivation of platform-specific semantics for latency

4 Examples of Real-time Streaming Networks

In the following, we show some applications of stream processing networks for real-time computing and also discuss specific issues implied by them which are relevant for modelling.

4.1 Fuel Injection

Figure 9 shows a grossly simplified model of a fuel injection system for internal combustion engines. Fuel injection is a real-time application with very strict timing requirements. Injecting the fuel to late or too early not only reduces the efficiency of the engine, but also increases the mechanical stress of the engine components, resulting in an acceptable outwear rate.

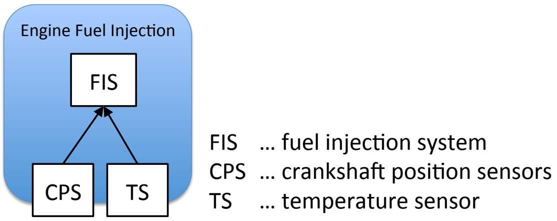

Engine Fuel Injection
FIS
CPS
TS
FIS ... fuel injection system
CPS ... crankshaft position sensors
TS ... temperature sensor

Fig. 9. Example of multi-rate system: Engine Fuel Injection

While any real fuel injection system is much more complex, our simplified model is sufficient to outline a relevant property when modelling it as a streaming network. The fuel injection system FIS collects inputs from two sources: the current motor temperature from sensor TS and the current crankshaft position from sensor CPS.

The important aspect of this simple model is that we have different rates involved here. The CPS sensor has to deliver its trigger signal in fixed coupling with the crankshaft position, every revolution of the crankshaft. Without the CPS trigger signal available, the fuel injection cannot operate. Also the latencies involved along processing the crankshaft position have to be precisely taken into account. In contrast, the motor temperature from sensor TS has much weaker requirements. Neither the rate nor the latency of that sensor are very significant, since the temperature of the engine changes relatively slowly during correct operation.

4.2 Car Platooning

A car platooning (CP) system is a technology to line up vehicles on a highway into virtual trains, automatically controlling distance and speed [1]. In this section we purely focus on the influence of CPS to the brake control of a car. We want to highlight the different levels of criticality when it comes to brake control in a modern car.

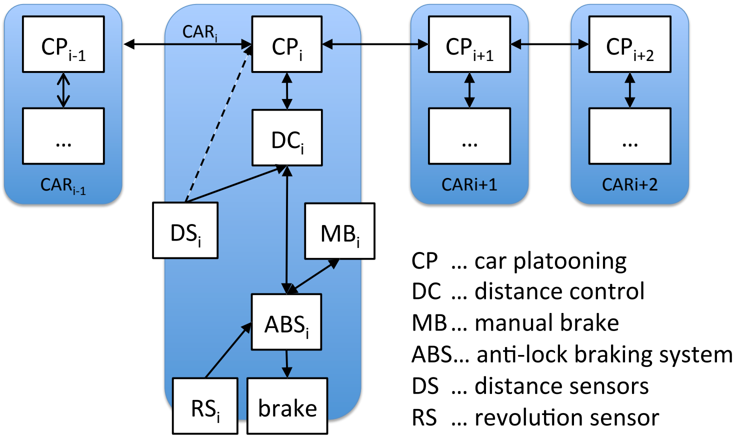


Fig. 10. Example of mixed criticalities: Car Platooning

Figure 10 shows the control chain of components in a modern car that can influence the activation of the brake. Quite standard nowadays is the anti-lock braking system (ABS) which has the highest control over the brake. The driver may activate the braking of the car with the manual brake (MB), but it is the ABS which finally decides when and how long the brake should be actually activated, giving priority to preserve steer-ability over short braking distance. To do so, ABS receives information about the current wheel revolution speed from sensor RS, and lowers the brake activation whenever the speed of a wheel drops. On top of MB acts the distance control (DC) system, which uses distance sensors (DS) in order to keep a minimum distance with other vehicles driving in front of the car. On top of DC may be a car platooning (CP) system which, to some extent, behaves similar to DC by taking DS into account in order to control the distance to the font car. CP and DC can actually share the input from the same distance sensors of the car. However, CP senses the environment beyond that, being in active communication with the neighbouring vehicles on the road, allowing smoother operation by starting distance adjustment measures before even a change was noticed via the DS sensors.

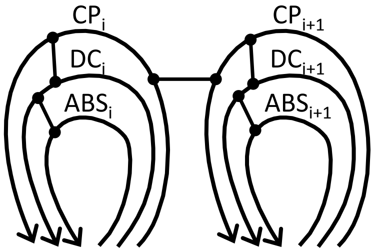


Fig. 11. Example of coupled control loops: Car Platooning

What this use case shows is that besides timing requirements there are also dependability requirements, resulting in different criticalities of the above services. The closer a component is to the brake, the higher is its criticality and the more control it has over the brake. Figure 11 visualises the different automatic control loops involved in that use case. Highest priority is given to ABS, as it is able to pause braking whenever it needs to do so in order to give priority to steer-ability. The manual break MB is not shown in that diagram, as it acts besides ABS since manual brake has to work even if ABS fails. DC has higher priority to CP regarding braking, since DC might detect a road obstacle while CP wants to accelerate in order to keep the distance with the vehicle driving in front of it.

This example demonstrates that in the system design, real-time requirements and criticality properties are orthogonal issues. Having the strongest real-time requirements has nothing to do with having the highest criticality in the system.

To summarise, real-time requirements are an important category of extra-functional properties, but there are other extra-functional requirements as well. So, scheduling resources may not simply be a real-time problem, but also a mixed-criticality problem.

5 Related Work

There are many approaches of modelling distributed systems with specification of extra-functional properties. In the following we present a sample of modelling approaches, including academic research and concrete tools.

An all-round modelling approach is the *Unified Modelling Language* (UML), which combines multiple modelling paradigms in a unified framework [12]. UML allows the modelling of a system at different abstraction levels. For example, with the UML *Use Case Diagram* (UCD) one can model the system application context without focusing on implementation details and formal interfaces. Regarding the application of UML to real-time systems there are different approaches. For example, *Real-Time UML* describes how to model resources, time, and concurrency [5]. The *UML Profile for Modeling and Analysis of Real-time and Embedded systems* (MARTE) extends Real-Time UML with concepts for timed processing and timed events, introducing also logical and physical clocks as different time sources [13, 3].

The UCD of UML provides an interesting concept of how to model system requirements at a high level, focusing on the different services to be provided. However, when using the mechanisms of UML and its extensions to specify, one also has to face the problems discussed in this paper.

An example of academic approaches of how to model timed systems is *Ptides*, a variant of the *Ptolemy* execution model [4, 2]. Ptides and Ptolemy have been developed by Ed Lee et al. at Berkeley. Ptides is a stream-based event processing model with support to model extra-functional properties and time sources for real-time processing. Ptides allows to model the processing chains on so-called platforms and the communication between multiple platforms, resulting in distributed systems. With Ptides, one can annotate the extra-functional properties of different components, like delays to locally receive or send a message. In addition, one can set the "due date" of messages, i.e., the time at which an output at an actuator should be produced. While this due date is initially set to the message creation time, it can then be incrementally increased by delay blocks along the processing path toward its final destination at an actuator. There are no inherent rules of where to increment the due date by how much, as long as the total delay of the multiple delay blocks along the processing chain add up to the desired time for the output to be produced. By this delay blocks one can obtain a timed system model that has a fixed semantics of the event timings, regardless of the underlying platform. From that point of view, Ptides is well-suited to model the latency of real-time systems in a platform-independent way via summative local latencies as described in Section 3. So far, the published research on Ptides did not address the issue of how to derive platform-specific latencies, which besides summative latencies, would also provide local absolute latencies. As such, Ptides provides a very useful summative timing semantics at platform-independent design model level, but cannot escape the challenge of how to obtain local absolute latencies, as discussed in Section 3, which would require a platform-specific design model.

As an example of a modelling tool with wide-spread industrial use is Matlab/Simulink from Mathworks [10]. In Simulink one can specify processing graphs with multiple update rates of the different components. Simulink is well-prepared for modelling multirate systems by a specific "Rate Transition" block [10]. The Rate Transition block can be parameterised in order to trade *data integrity* and *deterministic transfer* for faster response or lower memory requirements. Simulink's focus on update rates works relatively well to address the problem discussed in this paper, but only from the throughput point of view. Simulink does not provide the same flexibility for modelling event latency, as, for example, Ptides is able to offer.

Kirner and Maurer have recently introduced an interface specification for components of stream-based mixed-criticality systems [11]. This model includes the specification of the progress type of components (time-triggered or some form of event-triggered), but most importantly it offers an explicit way to specify trigger dependencies and trigger-decoupling of subsystems. They also introduced the classification of messages into event messages, state messages and semi-state messages in order to have a semantic justification of trigger coupling/decoupling [11]. Their mixed-criticality interface techniques could applied to other models like Ptides. While these mixed-criticality interfaces are useful to compose the timing behaviour from subsystem, this cannot completely remedy the timing specification problem discussed in this paper.

6 Summary and Conclusion

The specification of real-time properties in complex systems causes some challenges which we address in this paper. We have put our focus on streaming networks which are a well-suited design paradigm for cyber-physical systems with their omnipresent concurrent behaviour.

We have shown that it is not a straight forward process to specify real-time properties for streaming networks, neither at the requirements level nor at the system design level. More precisely, it is the specification of latency (or deadlines) that is not well suited for streaming networks, mostly because latency requirements are caused by the application environment and do not have a direct imprint at subsystem level. Throughput or processing rate on the other hand can be annotated to streaming networks relatively easy.

We resolve the situation by providing latency specifications with only summative semantics at the level of platform-independent design models. Using temporal behaviour analysis these latency specifications can be refined from a platform-specific design model into latency specifications, having both, a local absolute semantics as well as the original summative semantics at the global level. This result shows to what extent it is possible to specify platform-independent system design models with real-time semantics.

References

1. Carl Bergenhem, Henrik Pettersson, Erik Coelingh, Cristofer Englund, Steven Shladover, and Sadayuki Tsugawa. Overview of platooning systems. In *Proc. 19th ITS World Congress*, Vienna, Austria, Oct. 2012.
2. Janette Cardoso, Patricia Derler, John C. Eidson, Edward A. Lee, Slobodan Matic, Yang Zhou, and Jia Zou. *Systems Design, Modeling, and Simulation using Ptolemy II*, chapter Modeling Timed Systems, pages 355–393. Ptolemy.org, 1st edition, 2014. Available online at `http://ptolemy.org/books/Systems`.
3. Sebastien Demathieu, Frederic Thomas, Charles Andre, Sebastien Gerard, and Francois Terrier. First experiments using the UML profile for MARTE. In *Proc. 11the IEEE Int'l Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, pages 50–57, May 2008.
4. Patricia Derler, Thomas Huining Feng, Edward A. Lee, Slobodan Matic, Hiren D. Patel, Yang Zhao, and Jia Zou. PTIDES: A programming model for distributed real-time embedded systems. Technical Report UCB/EECS-2008-72, EECS Department, University of California, Berkeley, May 2008.

5. Bruce Powel Douglass. *Real Time UML: Advances in the UML for Real-Time Systems (3rd Edition)*. Addison-Wesley Professional, 3rd edition, Feb. 2004.
6. Stephen A. Edwards, Sungjun Kim, Edward A. Lee, Isaac Liu, Hiren D. Patel, and Martin Schoeberl. A disruptive computer design idea: Architectures with repeatable timing. In *Proc. IEEE International Conference on Computer Design*. IEEE, 2009.
7. Stephen A. Edwards and Edward A. Lee. The case for the precision timed (pret) machine. In *Proc. 44th Design Automation Conference (DAC)*, San Diego, California, Jun. 2007.
8. Hermann Kopetz. *Real-Time Systems - Design Principles for Distributed Embedded Applications*. Springer, 2nd edition, 2011. ISBN: 978-1-4419-8236-0.
9. Edward A. Lee. Cyber physical systems: Design challenges. In *Proc. 11th IEEE International Symposium on Object-oriented Real-time distributed Computing*, pages 363–369, Orlando, Florida, USA, May 2008.
10. The MathWorks Inc., Natick, Massachusetts, USA. *Simulink Reference*, r2015a edition, March 2015. Revised for Simulink 8.5 (Release 2015a).
11. Simon Maurer and Raimund Kirner. Cross-criticality interfaces for cyber-physical systems. In *Proc. 1st IEEE Int'l Conference on Event-based Control, Communication, and Signal Processing*, Krakow, Poland, June 2015.
12. OMG. *Unified Modeling Language: Superstructure (version 2.1.1)*. Object Management Group, Feb. 2007. OMG document number: formal/2007-02-05.
13. OMG. *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems*. Object Management Group, 1.1 edition, June 2011.

The Isabelle Refinement Framework
For Verification of Large Software Systems

Peter Lammich

TU Munich lammich@in.tum.de

Abstract. This paper overviews the techniques that we use to develop verified large software systems inside the Isabelle/HOL theorem prover. It is based on our development of the fully verified efficiently executable CAVA LTL model checker.

The verification follows a stepwise refinement approach, to which we adapted standard engineering techniques such as object orientation and modularization. It is entirely conducted in the Isabelle/HOL theorem prover, which results in a high confidence correctness theorem that only depends on the small inference kernel of Isabelle/HOL.

The techniques presented in this paper cover the Isabelle Refinement Framework, which provides a formalization of refinement calculus and a tool chain which makes it conveniently usable. Moreover, we describe the Isabelle Collection Framework, which provides an extensible library of efficient verified collection data structures. We also describe the object oriented techniques used to develop the automata library below our model checker, and the modularization techniques used to separate the various components of the model checker.

1 Introduction

The objective of this paper is to give an overview of the development techniques that we use to verify large-scale software systems in the Isabelle/HOL interactive theorem prover. We present some of the engineering techniques that we used to develop the verified CAVA model checker [7], a fully-fledged efficient LTL model checker.

Our development process is based on stepwise refinement, to which we adapt standard engineering techniques for structuring large software systems, like object orientation and modularization.

Stepwise refinement is a well-known technique to verify programs. The idea is to refine an abstract specification to an efficient implementation via a series of correctness preserving refinement steps. Usually, the first refinement steps introduce the algorithmic ideas of the program, and further refinement steps then replace the abstract data types used to describe the algorithm by efficient implementations.

Stepwise refinement reduces the proof complexity by separation of concerns: Instead of one big proof that deals with both, the high-level algorithmic ideas and the implementation details, it allows for several small proofs, each focusing on a single aspect. Our experience shows that direct correctness proofs of efficient implementations tend to get unmanageable already for medium-complex algorithms like Dijkstra's Shortest Paths.

Refinement calculus [2] formalizes stepwise refinement in a Hoare-logic like framework, based on rigorous mathematical foundations. Thus, it is well suited for a theorem prover based development.

Our main tool is the Isabelle Refinement Framework[20] (cf. Section 2), which implements a refinement calculus for shallowly embedded monadic programs. It comes with tool support, which makes it practically usable. Besides a verification condition generator, it also contains the Autoref tool [16], which can automatically refine abstract data types to efficient implementations. Suitable implementations are selected via user-adjustable heuristics.

When developing efficient algorithms, it is important to have a library of reusable general purpose data structures. The Isabelle Collection Framework [14] (Section 3) provides such a library. It is based on the concepts of interfaces, generic algorithms, and implementations. Its integration into Autoref ensures easy usability.

The CAVA model checker operates on different types of graphs and automata. To avoid redundancies, these are presented as a class diagram with inheritance. In Section 4, we give a brief overview of the CAVA Automata Library [17] and how it uses object oriented techniques inside Isabelle/HOL.

The CAVA model checker itself consists of several components, which are separately maintained and developed. In Section 5, we review the modularization techniques we use to ensure isolation between these components, and their interplay with verification.

Finally, we conclude the paper in Section 6.

Note that this paper is an overview paper, presenting results that have been detailed in [14, 20, 16, 17, 7], with some small parts of newer developments. We have tried to indicate new developments in the paper at the points where they are described. The main focus of this paper is on the refinement calculus and the associated tool chains. Further engineering techniques that helped us in developing the CAVA model checker are only briefly discussed, with references to more detailed descriptions.

1.1 Related Work

Refinement Based on Back et al.'s initial formalization in HOL [1], there are several formalizations of refinement calculus in different theorem provers (e. g. [4, 23, 3]). However, they usually focus on the meta-theory of refinement calculus, and only come with relatively small example programs that are actually verified.

For the Coq theorem prover, there is a refinement tool [5], which has been used to refine some algebraic algorithms to use efficient data structures. Also the Fiat-System [6] automatically synthesizes efficient implementations of database queries phrased in an abstract query language. Both tools resemble our Autoref-tool [16], which we use to automate canonical refinement steps.

Another implementation of data refinement is supported by the Isabelle Code Generator [9]. However, it relies on an extension of the code generator outside the logic. Moreover, it can only be used for deterministic algorithms, while many abstract algorithms of a model-checker are inherently nondeterministic.

Verification of Big Software Systems There are several verifications of big software systems, even bigger ones than the CAVA model checker. One example is the verified C compiler CompCert [21]. Here, modularization is achieved by splitting the compiler into several phases, which translate between different intermediate languages. For each translation step, bisimulation between the input and output is proved. Other important techniques include tailoring of algorithms to be verification friendly, and to use a-posteriori verification of results computed by external unverified algorithm.

Another big system that has been verified is the seL4 microkernel [12]. It uses a refinement-centric development process: First, a prototype of the kernel was implemented in Haskell. It serves both as an executable implementation that can be tested, and as a functional specification that can be reasoned about in a theorem prover. Then, an efficient C version was manually implemented, and proved to refine the Haskell prototype, which, in turn, was shown to satisfy the abstract specification. For the refinement proof between the C program and the Haskell prototype, the Autocorres tool [8] was developed. Similar to our approach, it implements a refinement calculus on shallowly embedded monadic programs. However, it features bottom-up refinement, i.e., a concrete program is abstracted, while our approach uses top-down refinement, where an abstract program is concretized.

2 Foundations of the Refinement Framework

The Isabelle Refinement Framework [20, 15] provides a refinement calculus [2] that is based on a nondeterminism monad [25]. It features a stepwise refinement based development approach, where an algorithm is first specified on an abstract level, and then refined towards an efficient implementation in possibly many correctness preserving steps.

Note that nondeterminism is essential for specifying abstract algorithms: For example, a standard textbook presentation of a workset algorithm might contain the operation ,,pick some element from the workset". However, a precise description of which element is picked is not possible until the data structure for the workset has been fixed. Thus, abstractly, one has to nondeterministically choose an element, and prove the algorithm correct for any choice.

In the remainder of this section, we briefly introduce the Isabelle Refinement Framework and its theoretical foundations.

2.1 The Refinement Monad

The Monadic Refinement Framework represents programs inside a monad over the type $'a\ nres = \textbf{res}\ 'a\ set \mid \textbf{fail}$. A *result* **res** X means that the program nondeterministically returns a value from the set X, and the result **fail** means that an assertion failed. The subset ordering is lifted to results as follows:

$$\textbf{res}\ X \leq \textbf{res}\ Y \equiv X \subseteq Y \ \mid\ \_ \leq \textbf{fail} \equiv \mathit{True} \ \mid\ \_ \leq \_ \equiv \mathit{False}$$

Intuitively, $m \leq m'$ (*m refines* m') means that all possible values of m are also possible values of m'. Note that this ordering yields a complete lattice on results, with smallest element **res** {} and greatest element **fail**. The monad operations **return** and **bind** (notation $\gg\!=$) are then defined as follows:

$$\textbf{return}\ x \equiv \textbf{res}\ \{x\}$$
$$\textbf{res}\ X \gg\!= f \equiv \mathit{Sup}\ \{f\ x \mid x{\in}X\} \ \mid\ \textbf{fail} \gg\!= f \equiv \textbf{fail}$$

Intuitively, **return** x is the result that contains the single value x, and $m \gg\!= f$ is sequential composition: Choose a value from m, and apply f to it.

As a shortcut to specify values satisfying a given predicate Φ, we define **spec** $\Phi \equiv$ **res** $\{x \mid \Phi\ x\}$. Moreover, we use a Haskell-like do-notation, and define a shortcut for assertions: **assert** $\Phi \equiv$ **if** Φ **then return** () **else fail**. Recursion is defined by a fixed point: **rec** $B\ x \equiv$ **do** {**assert** (*mono B*); *gfp B x*}. As we use the greatest fixed point, a non-terminating recursion causes the result to be **fail**. This matches the notion of total correctness. We assert monotonicity of the function's body. Note that the standard way of defining recursion is w.r.t. a flat ordering of results, where **fail** is the top element. Thus, we require monotonicity w.r.t. both, the refinement ordering and the flat ordering, in which case the greatest fixed points coincide. Note that monotonicity w.r.t. both orderings follows by construction [13] for any program that only uses the monad combinators.

On top of the **rec** primitive, we define loop constructs like **while** and **foreach**, with an explicit state threaded through the loop.

2.2 Data Refinement

In a typical refinement based development, one also wants to refine the representation of data.

A data refinement is specified by a *refinement relation* between concrete and abstract values. In many cases, this relation is single-valued, and can be expressed by an *abstraction function* from concrete to abstract values and an invariant on concrete values. Note, however, that refinement relations typically are neither left nor right total.

A prototypical example is implementing sets by *distinct lists*, i.e. lists that contain no duplicate elements. Here, the refinement relation $\langle R\rangle\mathit{list\_set\_rel}$ relates a distinct[1] list to the set of its elements, where the elements are related by R. This relation is not left-total, as lists with duplicate elements have no abstract counterpart. This reflects the concrete data structure's invariant. Also, this relation is not right-total, as infinite sets cannot be implemented by lists.

[1] Assuming R is single-valued

Given a refinement relation R, we define the function $\Downarrow R$ to map results over the abstract type to results over the concrete type:

$$\Downarrow R\ (\mathbf{res}\ A) \equiv \mathbf{res}\ \{c \mid \exists a \in A.\ (c,a) \in R\} \quad | \quad \Downarrow R\ \mathbf{fail} \equiv \mathbf{fail}$$

Intuitively, $\Downarrow R\ m_2$ is the largest concrete result, such that all its values have abstract counterparts in m_2. Thus, $m_1 \leq \Downarrow R\ m_2$ (notation $m_1 \leq_R m_2$) states that m_1 is a refinement of m_2 w. r. t. the refinement relation R, i. e. all concrete values in m_1 correspond to abstract values in m_2.

Note that we originally [20] defined the refinement relation differently: By only including concrete elements for which all abstractions are contained in the abstract result, we made $\Downarrow_R$ an adjoint of a Galois connection, which seemed theoretically beautiful at first glance. However, with this definition, we could prove some refinement rules only for single valued relations. However, during our development of a DFS framework [19], we also required (non-single valued) projection relations to reason with ghost variables. Thus, we changed the definitions to better generalize to arbitrary relations, at the cost of loosing the Galois connection property, which required reworking some proofs.

2.3 Refinement Calculus

For each combinator of the nres-monad, we define two refinement rules. One for *specification refinement*, which proves properties of the form $m \leq \mathbf{spec}\ \Phi$, and one for *pure data refinement*, which proves properties of the form $m \leq \Downarrow R\ m'$, where m and m' have the same top-level combinator. Intuitively, a specification refinement replaces an abstract specification by an algorithmic implementation (e. g. „Some path from u to v” by a depth-first search algorithm), and a pure data refinement replaces abstract types by concrete data structures (e. g. set by distinct list). For example, the rules for **return** and $\gg=$ are the following:

$$\Phi\ x \Longrightarrow \mathbf{return}\ x \leq \mathbf{spec}\ \Phi$$
$$(x,\ x') \in R \Longrightarrow \mathbf{return}\ x \leq_R (\mathbf{return}\ x')$$

$$m \leq \mathbf{spec}\ (\lambda x.\ f\ x \leq \mathbf{spec}\ \Phi) \Longrightarrow m \gg= f \leq \mathbf{spec}\ \Phi$$
$$[\![m \leq_{R'} m';\ \bigwedge x\ x'.\ (x,\ x') \in R' \Longrightarrow f\ x \leq_R (f'\ x')]\!] \Longrightarrow m \gg= f \leq_R m' \gg= f'$$

Consider a refinement goal of the form $m \leq_R m'$. If the programs are similar enough, i.e., they have the same structure, where m may contain an arbitrary expression at places where m' contains a **spec** and the refinement relation is Id (note that $\forall\ m.\ \Downarrow Id\ m = m$), resolution with the refinement rules leaves us with *verification conditions* over the basic operations in the program. The Isabelle Refinement Framework comes with a verification condition generator (VCG), which automates this process, and has some additional rules to tolerate certain structural changes.

2.4 Refinement Based Algorithm Development

In a typical development based on stepwise refinement, one specifies a series of programs $m_1 \geq \ldots \geq m_n$, such that m_1 has the form **assert** *pre;* **spec** *post*, and m_n is the final implementation. In each refinement step (from m_i to m_{i+1}), some aspects of the program are refined.

Refinement is modular, i.e., one can prove refinements for parts of a program in isolation. This is important for having libraries of standard algorithms, which can be used in the program to be developed. One such example is the Isabelle Collection Framework (cf. Section 3). Also, it allows to independently develop the components of larger programs, as we illustrate in Section 5.

Example 1. Given a finite set S of sets, the following specifies a set r that contains at least one element from every non-empty set in S:

$$\mathit{sel}_1\ S \equiv \mathbf{do}\ \{\mathbf{assert}\ (\mathit{finite}\ S);\ (\mathbf{spec}\ r.\ \forall s \in S.\ s \neq \{\} \longrightarrow r \cap s \neq \{\})\}$$

This specification can be implemented by iteration over the outer set. In each iteration step, the result set must not shrink, and it must contain an element from the current inner set, if this is not empty.

```
sel2 S ≡ do {
  assert (finite S);
  foreach S (λs r. spec r'. r' ⊇ r ∧ (s ≠ {} ⟶ r' ∩ s ≠ {})) {}
}
```

Using the verification condition generator, it is straightforward to show that sel_2 is a refinement of sel_1:

```
lemma sel2 S ≤ sel1 S
  unfolding sel2_def sel1_def
  by (refine_vcg foreach_rule[where I=λit r. ∀s∈S−it. s≠{} ⟶ r∩s≠{}])
    auto
```

Note that the invariant for the foreach-loop is explicitly specified here. It is parametrized over the set *it* of elements still to be iterated over, and the current state r of the loop.

Next, we want to further refine the program: In each iteration, we want to pick an arbitrary element from the current inner set, and add it to the result set. We specify the new algorithm:

```
sel3 S ≡ do {
  assert (finite S);
  foreach S (λs r.
    if s={} then return r
    else do {
      x←spec x. x∈s;
      return (insert x r)
    }
  ) {}
}
```

Note that only the body of the foreach-loop has changed. Using the VCG, it is straightforward to show that this algorithm refines the previous one:

```
lemma sel3 S ≤ sel2 S
  unfolding sel3_def sel2_def by (rule refine_IdD,refine_vcg inj_on_id) auto
```

Now assume that finding a representative element from a set is hard. Thus, every inner set comes with an pre-computed representative. We define a refinement relation between sets of sets with representatives, and sets of sets:

```
    definition repr_set_rel ≡ {(S',S).
(*1*) S = snd`S'
(*2*) ∧ (∀(b,s)∈S'. case b of None ⇒ s={} | Some x ⇒ x∈s)
(*3*) ∧ (single_valued (S'\<inverse>))
      }
```

Proposition (1) ensures that the abstract set can be obtained from the concrete set by projecting away the representatives. Proposition (2) ensures that the attached representatives are actual representatives, where an option-type is used to have *None* as representative for the empty set. Finally, proposition (3) ensures that we do not add more than one representative for each set. This is important to ensure that a finite abstract set must be represented by a finite concrete set, over which iteration is well-defined.

Finally, we phrase the refined algorithm, and prove refinement:

```
definition sel4 S ≡ do {
  assert (finite S);
  foreach S (λ(b,_) r.
    case b of None ⇒ return r | Some x ⇒ return (insert x r)
  ) {}
}

lemma (S',S)∈repr_set_rel ⟹ sel4x S' ≤ sel3 S
  unfolding sel4x_def sel3_def
  apply (rule refine_IdD)
  apply (refine_rcg FOREACH_refine_rcg[where α=snd])
  [...] (* Omitted 8 lines of standard Isabelle text to prove the VCs *)
  done
```

2.5 Automatic Refinement

Many refinements, which are typically performed at the end of a refinement based development, are pure data refinements, i. e. the overall structure of the program is preserved, and only some abstract types are refined to concrete data structures. Given which abstract types to refine to which concrete data structures, as well as refinement rules for the required operations, the concrete program and the refinement theorem can be automatically synthesized from the abstract program.

We have implemented such a synthesis procedure in the Autoref tool [16]. It is based on the idea to express data refinement by relators [24].

It contains various heuristics to automatically select appropriate data structures and algorithms for the types and operations in the abstract program. The most important ones are the homogeneity principle and priorities. The homogeneity principle intuitively states that the result of an operation should be implemented by the same data structure as the operands. This avoids frequent casts between different implementations, thus producing a cleaner and more predictable synthesis result. Priorities can be assigned to both, data structures and algorithms. They are used to prefer efficient data structures and algorithms over less efficient ones.

Moreover, Autoref supports instantiation of generic algorithms via recursive synthesis. A generic algorithm implements an operation in terms of other operations. For example, union of finite sets may be implemented by iterating over one set, and inserting its elements into the other. When Autoref encounters a union operation, and decides to use this generic algorithm, it will try to synthesize algorithms for iteration and insertion.

Using priorities, generic algorithms may be specialized. For example, there is a more efficient union-operation on red-black trees. Its rule has a higher priority than the generic algorithm, such that Autoref will try it first. Similarly, if one can prove that the sets to be joined are disjoint, union on distinct lists can be efficiently implemented by concatenation. This rule depends on an additional side condition, which our tool will try to prove using some standard Isabelle tactics. If the proof fails, the generic algorithm is used.

2.6 Code Generation

Once the program is refined to a deterministic program that only uses executable constructs, we have to generate actual code from it. This is done in two steps: In the first step, the program is transfered to a deterministic monad, and in the second step, it is translated to source code of an actual programming language.

Transfer to Deterministic Program The combinators of the nres-monad itself are defined using non-executable constructs. For execution, we define the dres-monad over the type $'a\ dres = dsucceed \mid dreturn\ 'a \mid dfail$.

The function $nres\_of :: 'a\ dres \Rightarrow 'a\ nres$ maps a result from the dres-monad to its corresponding result from the nres-monad. Given a deterministic program m in the nres-monad, it is straightforward to transport it to the dres-monad, i.e., automatically synthesize a program m' with $nres\_of\ m' \leq m$. Moreover, if m is tail recursive (i.e. does not contain the **rec** combinator), it can be transported to a plain HOL expression. That is, we can automatically synthesize a term m'' with **return** $m'' \leq m$.

Isabelle's Code Generator When the program is refined to the dres-monad or to a plain expression, and all functions used by the program are executable (i.e., the code generator knows how to generate code for them), the code generator of Isabelle/HOL [10] can be used to generate code in one of its supported languages, which are currently SML, OCaml, Scala, and Haskell. Note that code generation happens outside the logic of Isabelle/HOL, and thus belongs to the trusted code base. However, there is a pen-and-paper proof of its correctness [10].

Example 2. Reconsider the program from Example 1. We want to implement the input by a distinct list of distinct lists. As retrieving a representative element from a non-empty list is simple, let's drop our last refinement step and start at program sel_3 again.

As Autoref is often applied in the last refinement step before code generation, it can combine the data refinement and the transportation to the dres-monad or plain expression. Thus, an executable version of sel_3 is generated as follows:

```
schematic_lemma sel4'_aux:
  assumes [autoref_rules]: (Si,S)∈⟨⟨Id⟩list_set_rel⟩list_set_rel
  shows (?c::?'c,sel3 S)∈?R
  unfolding sel3_def by (autoref_monadic (plain))
concrete_definition sel4' uses sel4'_aux
prepare_code_thms sel4'_def
export_code sel4' in SML
```

Here, the **assumes**-line is an annotation that the parameter S should be refined by a list of lists. The relation $?R$ for the result type is left unspecified. Autoref also decides to use a distinct list, as it knows nothing about the (polymorphic) element type, and thus cannot derive an ordering or hash function, which would be required for more efficient data structures. The (plain) option indicates to transfer to a plain function, instead of the default transfer to the dres-monad. Finally, the **concrete_definition** command extracts the concrete program from the refinement theorem and names it sel_4'. The last two lines then generate the following SML-code:

```
fun sel4' A_ si =
  Foldi.foldli si (fn _ ⇒ true)
    (fn x ⇒ fn sigma ⇒
      (if Autoref_Bindings_HOL.is_Nil x then sigma
        else let
               val xa = List.hd x;
             in
               Impl_List_Set.glist_insert (HOL.eq A_) xa sigma
             end))
    [];
```

The code has the same structure as the original program. The foreach-loop has been replaced by a fold function[2], and the set operations have been replaced by corresponding list operations. The extra parameter $A_{\_}$ contains the equality operation on the polymorphic element type, which is required by the insert operation.

[2] The variant *foldli* has an additional break condition, which, however its not used here, and thus set to **fn** _ ⇒ *true*.

As the generated code lives outside the logic of Isabelle, we cannot prove that it coincides with $sel_4{}'$. However, by chaining all the refinement theorems we have obtained on our way from the specification sel_1 down to the executable version $sel_4{}'$, we can prove that $sel_4{}'$ is actually correct w. r. t. the specification:

$$(S',S) \in \langle\langle Id\rangle list\_set\_rel\rangle list\_set\_rel \Longrightarrow \forall s{\in}S{-}\{\{\}\}.\ set\ (sel_4{}'\ S') \cap s \neq \{\}$$

3 The Isabelle Collection Framework

Having a library of re-usable standard data structures greatly reduces the effort required to produce efficient implementations. In this section, we briefly describe the Isabelle Collection Framework (ICF), which provides such a library.

It is seamlessly integrated into Autoref, such that many collection data structures are readily available, without any further setup. The current ICF is a defacto reimplementation of the original framework [14], to support nested data structures (e.g. distinct lists of distinct lists), and make use of the Autoref tool to instantiate generic algorithms.

The ICF is based on the concepts of interfaces, generic algorithms, and implementations. Its main features are easy usability and extensibility, which is achieved through seamless integration into the Autoref tool: Its heuristics select appropriate data structures that the user do not even have to know about. Moreover, new interfaces, generic algorithms, and implementations can be added to the ICF easily and without changing the original code base.

3.1 Interfaces

An interface describes an abstract data type and the operations on it. The default interfaces which come with the ICF are map, set, priority queue, and list. All the interfaces come with a large set of pre-defined operations, and the setup required for Autoref to identify those operations in the abstract program.

For example, the map interface comes with an emptiness check operation, and the abstract expressions $m = Map.empty$ and $dom\ m = \{\}$ may be identified as emptiness check by Autoref.

3.2 Generic Algorithms

The ICF heavily relies on generic algorithms as a tool to avoid code duplication and allow rapid prototyping of new data structures. For example, the ICF has generic algorithms to derive most operations on (finite) maps from five basic operations: empty-map, lookup, update, remove, and fold. Moreover, it has generic algorithms to derive a set implementation from a map implementation, by instantiating the value type to unit. This allows for rapid prototyping of a new data structure, as all operations on sets and maps become available once one has implemented the five basic map operations. Moreover, in many cases the generic algorithms are reasonably efficient and match the default implementation of the operation for this data structure. This way, code duplication is avoided, as the generic algorithm is shared between many data structures. If a data structure supports a more efficient version of an operation, specialization is used to override the generic algorithm.

3.3 Implementations

An implementation provides a concrete data structure for an interface. It consists of a refinement relation and implementations of some of the operations, along with their correctness lemmas.

Note that an implementation needs not provide all operations. Some of the operations may be filled in by generic algorithms, and others may not be supported at all. The Autoref tool will only select implementations that support all operations required by the program to be refined.

Available Implementations Examples for data structures provided by the ICF are red-black trees and hash tables for sets and maps, distinct lists for sets, association lists for maps, characteristic functions for sets, bit-vectors for (dense) sets of natural numbers, and arrays for (dense) maps from natural numbers.

While the red-black tree and list based data structures are purely functional, hash-tables, bit-vectors, and arrays are based on mutable arrays with undo-history (called DiffArray in Haskell) which behave like functional arrays, but use destructive update internally.

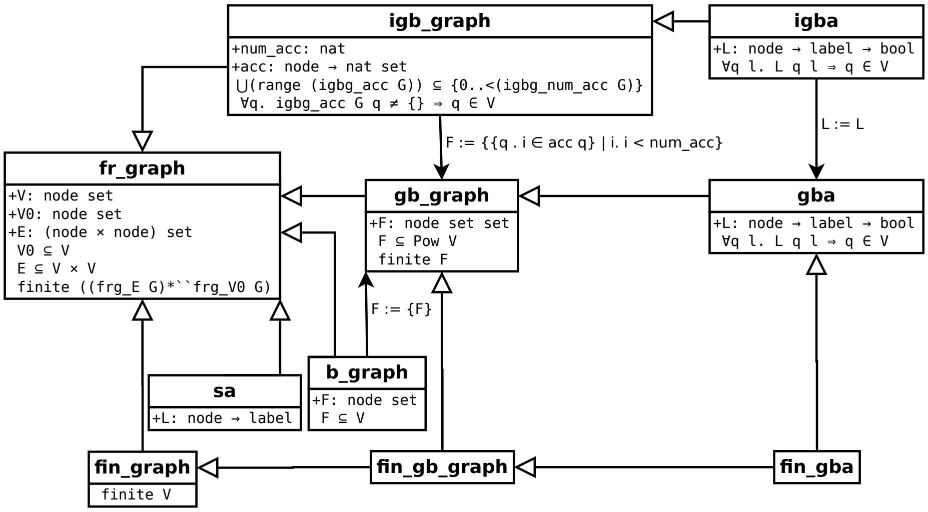
For those arrays, access to the latest version is always efficient, while access to earlier versions gets more expensive as older the accessed version is. However, many algorithms access their data in a linear fashion, and for linear access, the array-based implementations are considerably faster than purely functional implementation.

One drawback is that the mutable arrays with undo-history have to be implemented outside the logic, and thus contribute to the trusted code base. In [18], we presented an alternative approach that allows to reason about imperative features inside the logic.

4 The CAVA Automata Library

While the ICF organizes abstract types and their implementations, it has only limited support to establish a hierarchy on the interfaces: Type classes can be used to define specialized interfaces, which support additional operations: For example, the interface *ordered-set* constrains its elements to be in a linear order type class, and then provides additional operations like minimum.

When we developed the CAVA Automata Library [17], which formalizes the various graph and automata types that occur in the CAVA Model Checker, we realized that there are many redundancies between the various types, which we eliminated by structuring them in a class hierarchy:



Each class inherits the fields and invariants of its base classes, and may add new fields and invariants. Moreover, some of the classes may be specializations of other classes, as indicated by solid arrows. For example, Büchi automata can be seen as generalized Büchi automata with a single acceptance class, as indicated by the solid arrow from class *b_graph* to *gb_graph*.

Internally, classes are implemented by a mixture of locales [11] and records [22]. The records provide a mechanism to declare the fields of the classes, and exploit polymorphism to have subtyping, i.e., the type of a base class matches on the type of its subclasses. However, they are restricted to single inheritance, which was not a problem for our design[3].

Locales provide a mechanism to capture the invariants of a class. Moreover, inside a class' locale, concepts can be defined and theorems can be proven, which are inherited to the subclasses. For example, the class *fr_graph* defines the concept of a path between two nodes, and proves theorems about it. These are available in all subclasses.

Methods with static binding correspond to functions that take a parameter of a class' record type. Inside such a method, we may re-use the corresponding method from the superclass. For example, renaming the states of an automaton is implemented as first renaming the nodes of the underlying graph, and then renaming the set of accepting states.

Definition of methods with dynamic binding (i. e. virtual methods) is more tricky. We avoided this in our Automata library, and leave it to future research to evaluate the different possible approaches.

Implementation is done via the Autoref-Tool, defining the classes as abstract types, and relating them with implementations. The implementations are also

[3] Note that we actually support multiple inheritance as long as all the fields can be added by following a single path up the hierarchy.

structured via records, such that implementations of base classes may be reused to implement subclasses. For example, a *gb_graph* may be implemented by augmenting an implementation of an *fr_graph* with an acceptance set.

5 The CAVA LTL Model Checker

Figure 1 shows the overall architecture of CAVA. It follows a standard approach for LTL model checkers: The input is an LTL formula and a model, which is described either as a while program over Boolean variables or in Promela, the modeling language of SPIN. The model is converted to a Kripke structure, i. e. a directed graph with sets of atomic propositions annotated at the nodes.

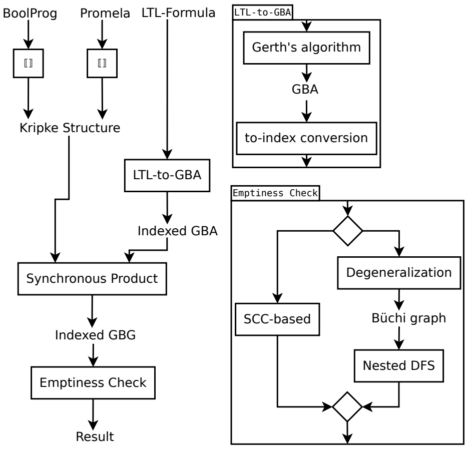


Fig. 1: Structure of the CAVA Model Checker

The LTL formula is converted to a generalized Büchi automaton, which accepts all infinite words that do *not* satisfy the formula.

Then, the synchronous product of the Kripke structure and the generalized Büchi automaton is computed, resulting in an generalized Büchi graph. Finally, the generalized Büchi graph is checked for emptiness by either using a strongly connected component algorithm, or by degeneralizing it and using nested depth first search. The result of the emptiness check either declares the automaton as empty, in which case the model satisfies the formula, or it returns a counterexample, which is a representation of an infinite run of the model that violates the formula.

The different components of CAVA are implemented and maintained by different developers. Thus, it is important to decouple them as much as possible. The interfaces between the components are based on the classes of the CAVA Automata Library.

The components are linked on two levels: the specification level and the implementation level. The specification level describes the abstract components' effect on the abstract automata data structures, using nondeterminism to leave room for different implementations. For example, the result of the intersection may be any automaton whose language is the intersection of the Büchi-automata's language with the system's runs.

The link at the implementation level is realized as generic algorithm: Given consistent implementations of the components which satisfy their specifications, a model-checker is constructed and proven correct. To obtain the actual model-checker and correctness proof, the generic algorithm is instantiated with the actual implementations.

These are the only points where the different components of the model checker are connected. Thus, changes to the components remain local, and do not affect the rest of the system. This greatly increases the maintainability and extensibility of the system. For example, we added the SCC-based emptiness check algorithm to the system later. After formalizing and proving the new algorithm correct, we could simply replace the original emptiness check component by a dispatcher component, which selects the algorithm based on a flag.

6 Conclusion

We have presented an infrastructure to develop large-scale verified software systems. It is based on stepwise refinement, which reduces proof complexity by splitting the correctness proof into independent parts. Our verification process is done entirely inside the Isabelle/HOL theorem prover. Thus, our correctness theorems only depend on the small inference kernel of Isabelle/HOL, which gives them a very high confidence. The user of our framework is supported by a tool chain which simplifies the proving process by automating canonical tasks.

Using the fully verified CAVA LTL model checker as a case study, we have shown how to adapt standard engineering techniques like object orientation and modularization to our development process.

References

1. Back, R.J.R., von Wright, J.: Refinement concepts formalized in higher order logic. Formal Aspects of Computing 2 (1990)
2. Back, R.J., von Wright, J.: Refinement Calculus — A Systematic Introduction. Springer (1998)
3. Boulmé, S.: Intuitionistic refinement calculus. In: Typed Lambda Calculi and Applications, LNCS, vol. 4583, pp. 54–69. Springer (2007)
4. Butler, M., Långbacka, T.: Program derivation using the refinement calculator. In: TPHOLs, LNCS, vol. 1125, pp. 93–108. Springer (1996)
5. Cohen, C., Dénès, M., Mörtberg, A.: Refinements for free! In: Gonthier, G., Norrish, M. (eds.) CPP, LNCS, vol. 8307, pp. 147–162. Springer (2013)
6. Delaware, B., Pit-Claudel, C., Gross, J., Chlipala, A.: Fiat: Deductive synthesis of abstract data types in a proof assistant. SIGPLAN Not. 50(1), 689–700 (Jan 2015), http://doi.acm.org/10.1145/2775051.2677006
7. Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.G.: A fully verified executable LTL model checker. In: CAV, LNCS, vol. 8044, pp. 463–478. Springer (2013)
8. Greenaway, D., Lim, J., Andronick, J., Klein, G.: Don't sweat the small stuff: Formal verification of C code without the pain. In: PLDI. pp. 429–439. ACM (jun 2014)
9. Haftmann, F., Krauss, A., Kunčar, O., Nipkow, T.: Data refinement in isabelle/hol. In: ITP, LNCS, vol. 7998, pp. 100–115. Springer (2013)

10. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Functional and Logic Programming (FLOPS 2010). LNCS, Springer (2010)
11. Kammüller, F., Wenzel, M., Paulson, L.C.: Locales - A sectioning concept for isabelle. In: TPHOLs. pp. 149–166 (1999)
12. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an OS kernel. In: Matthews, J.N., Anderson, T.E. (eds.) Proc. ACM Symp. Operating Systems Principles. pp. 207–220. ACM (2009)
13. Krauss, A.: Recursive definitions of monadic functions. In: PAR. vol. 43, pp. 1–13 (2010)
14. Lammich, P., Lochbihler, A.: The Isabelle Collections Framework. In: ITP. LNCS, vol. 6172, pp. 339–354. Springer (2010)
15. Lammich, P.: Refinement for monadic programs. In: Archive of Formal Proofs. http://afp.sf.net/entries/Refine_Monadic.shtml (2012), formal proof development
16. Lammich, P.: Automatic data refinement. In: Interactive Theorem Proving, LNCS, vol. 7998, pp. 84–99. Springer Berlin Heidelberg (2013)
17. Lammich, P.: The CAVA automata library. In: Isabelle Workshop (2014)
18. Lammich, P.: Refinement to imperative/hol. In: Interactive Theorem Proving, LNCS, vol. 9236, pp. 84–99. Springer (2015)
19. Lammich, P., Neumann, R.: A framework for verifying depth-first search algorithms. In: CPP. pp. 137–146 (2015)
20. Lammich, P., Tuerk, T.: Applying data refinement for monadic programs to Hopcroft's algorithm. In: ITP. LNCS, vol. 7406, pp. 166–182. Springer (2012)
21. Leroy, X.: A formally verified compiler back-end. J. Automated Reasoning 43, 363–446 (2009)
22. Naraschewski, W., Wenzel, M.: Object-oriented verification based on record subtyping in higher-order logic. In: TPHOLs. pp. 349–366 (1998)
23. Preoteasa, V., Back, R.J.: Invariant diagrams with data refinement. Formal Aspects of Computing 24(1), 67–95 (2012)
24. Reynolds, J.C.: Types, abstraction and parametric polymorphism. In: IFIP Congress. pp. 513–523 (1983)
25. Wadler, P.: Comprehending monads. In: Mathematical Structures in Computer Science. pp. 61–78 (1992)

SAC Goes Cluster

From functional array programming to distributed memory array processing

Thomas Macht[1,2] and Clemens Grelck[2]

[1] VU University Amsterdam
De Boelelaan 1105, 1081 HV Amsterdam, Netherlands
t.macht@student.vu.nl
[2] University of Amsterdam
Informatics Institute
Science Park 904, 1098 XH Amsterdam, Netherlands
c.grelck@uva.nl

Abstract. SAC (Single Assignment C) is a purely functional, data-parallel array programming language that predmoninantly targets compute-intensive applications. Thus, clusters of workstations, or more generally distributed address space supercomputers, form an attractive compilation target. Notwithstanding, SAC today only supports shared address space architectures, graphics accelerators and heterogeneous combinations thereof.

In our current work we aim at closing this gap. At the same time we are determined to uphold SAC's promise of entirely compiler-directed exploitation of concurrency, no matter what the target architecture is. It is well known that distributed memory architectures are going to make this promise a particular challenge.

Despite SAC's functional semantics, it is generally not straightforward to infer exact communication patterns from memory architecture agnostic code. Therefore, we intend to capitalise on recent advances in network technology, namely the closing of the gap between memory bandwidth and network bandwidth. We aim at a solution based on an implementation of software distributed shared memory (SDSM) and large per-node software-managed cache memories. To this effect the functional nature of SAC with its write-once/read-only arrays provides a strategic advantage that we aim to exploit.

Throughout the paper we further motivate our approach, sketch out our implementation strategy and show preliminary experimental evaluation.

1 Introduction

Single Assignment C (SAC) [9] is a functional data parallel language specialised in array programming. The goal of the language is to combine high productivity programming with efficient parallel execution. Data parallelism in SAC is based on array comprehensions in the form of `with`-loops that are used to create immutable arrays and to perform reduction operations. At this point, we can compile SAC source code into data parallel programs for shared memory architectures, CUDA-enabled graphics accelerators including hybrid systems and the MicroGrid architecture. However, the SAC compiler and runtime system do not yet support symmetric distributed memory architectures like clusters.

Our goal is to add efficient support for distributed memory architectures to the SAC compiler and runtime system. We aim to achieve competitive speedups for high-performance computing applications.

In a shared memory system, all nodes share a common address space. By contrast, in a distributed memory system, each node has a separate address space. In order to access remote data in a distributed memory programming model, the programmer must be aware of the data item's location and use explicit communication. While distributed memory systems can scale to greater size, the shared memory model is simpler to program. Distributed Shared Memory (DSM) aims to combine both models; it provides a shared memory abstraction on top of a distributed memory architecture. DSM can be realised in software or in hardware; hybrid solutions also exist. Partitioned Global Address Space (PGAS) is a programming model that lies in between the local and global view programming models. PGAS logically partitions a global address space such that a portion of it is local to each process, thereby exploiting locality of reference. PGAS is the underlying model of programming languages like Chapel [5].

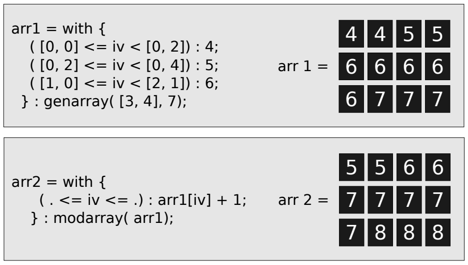
In the remainder of this paper we will first give an introduction to the SAC language and then motivate our current work, SAC for clusters. Subsequently, we will discuss our implementation and show preliminary performance results.

2 Single Assignment C

Single Assignment C (SAC) is a data parallel language for multi- and many-core architectures. For an introduction to SAC see [9]. The language aims to combine the productivity of high-level programming languages with the performance of hand-parallelized C or Fortran code. As the name suggests, the syntax is inspired by C. Other than C, however, SAC is a functional programming language without side-effects.

SAC is specialised in array programming; it provides multi-dimensional arrays that can be programmed in a shape-independent manner. While the language only includes the most basic array operations it comes with a comprehensive library. Conceptually, SAC's functional semantics requires to copy the full array whenever a single element is updated. To minimise the resulting overhead, SAC uses reference counting. This facilitates in-place updates of data structures when they are no longer referenced elsewhere. See [12] for SAC's memory management.

Array operations are typically implemented by `with`-loops, a type of array comprehension, which comes in three variants. See Figure 1 for examples. Both `genarray` and `modarray` `with`-loops create an array; `modarray` does so based on an existing array. For individual indices or sets of indices, expressions define the value of the corresponding array element(s). Independently for each index, the associated expression is evaluated and the corresponding array element is initialised. The third `with`-loop variant, `fold`, performs a reduction operation over an index set. As we have do not distribute these `with`-loops, we will not discuss them in this paper.



```
arr1 = with {
    ( [0, 0] <= iv < [0, 2]) : 4;
    ( [0, 2] <= iv < [0, 4]) : 5;
    ( [1, 0] <= iv < [2, 1]) : 6;
  } : genarray( [3, 4], 7);
```

```
         4 4 5 5
arr 1 =  6 6 6 6
         6 7 7 7
```

```
arr2 = with {
      ( . <= iv <= .) : arr1[iv] + 1;
    } : modarray( arr1);
```

```
         5 5 6 6
arr 2 =  7 7 7 7
         7 8 8 8
```

Fig. 1: Examples of genarray and modarray `with`-loops and resulting arrays

All variants of `with`-loops have in common that the compiler may evaluate individual expressions independently of each other in any order and that write-accesses are very restricted. These properties allow us to parallelise `with`-loops in an efficient way. While `with`-loops denote opportunities for parallelism, the decision whether they are actually executed in parallel or not is taken by the compiler and runtime system. At all times, program execution is either sequential or a `with`-loop is processed in parallel.

The SAC compiler is a many-pass compiler and emits platform-specific C code. The compiler spends a lot of effort on combining and optimising `with`-loops [10]. Currently, the compiler includes backends for symmetric multi-cores [8], GPUs (based on CUDA) [14] and the MicroGrid many-core architecture [15]. Heterogeneous systems are supported as well [6] and there have been experiments with OpenMP as a compilation target.

3 Motivation

In this section we argue why it is useful to add support for distributed memory architectures to SAC, why we followed a software DSM-based approach and why we decided to build a custom compiler-integrated DSM system.

3.1 Why support distributed memory architectures?

Distributed memory architectures are more cost-efficient, more scalable, and distributed memory architectures dominate high-performance computing. Currently, 86% of the TOP500 supercomputers are clusters and 14% have a Massively Parallel Processing (MPP) architecture [1] which is also a type of distributed memory system. While they are still predominant in commodity hardware, typical shared memory architectures have long vanished from the TOP500 list: single processors by 1997 and Symmetric multiprocessing (SMP) architectures by 2003 [1].

Message passing, and in particular MPI, is still the prevailing programming model for distributed memory systems [7]. While such a local view or fragmented programming model meets the performance requirements, it lacks programmability [5]. The programmer is responsible for the decomposition and distribution of data structures. Algorithms operate on the local portion of data structures and require explicit communication to access remote data. Data distribution and communication statements obscure the core algorithm.

By contrast, global view programming represents a higher-level alternative. In this model, the programmer works with whole data structures and writes algorithms that operate on these whole data structures. Data transfers and work distribution are handled implicitly. The algorithm is specified as a whole and not interleaved with communication. SAC offers a global view of computation to the programmer. By adding support for distributed memory architectures to SAC, we can utilise its global view programming model to make programming for distributed memory systems more efficient.

3.2 Why a software DSM-based solution?

Distributed Shared Memory (DSM) provides a shared memory abstraction on top of a physically distributed memory. An overview of issues of Distributed Shared Memory (DSM) systems can be found in [17]. DSM can be realised in software or hardware; hybrid systems also exist. In the context of this work, we focus on software solutions. According to [19], the first software DSM system was Ivy which appeared in 1984. Until the early 1990's, several other software DSM systems were proposed. Examples include Linda, Munin and Shiva [17].

These early DSM systems have not been adopted on a large scale due to shortcomings in performance. Explicit message passing, and in particular MPI, remain the predominant programming model for clusters. However, Bharath et al. suggest that it is time to revisit DSM systems [18]. They argue that early DSM systems were not successful because of slow network connections at the time. In the meantime, the picture has changed. Network bandwidth is comparable to main memory bandwidth and network latency is only one order of magnitude worse than main memory latency. According to Bharath et al. these developments reduce DSM to a cache management problem. They propose to use the improved network bandwidth to hide latency. As we will discuss in Section 4, our implementation uses that trick as well.

3.3 Why a custom DSM system?

In order to support distributed memory systems, we could run a SAC program on top of an existing software DSM system. Instead, we decided to integrate a custom DSM system into the SAC compiler and runtime system. This allows us to exploit SAC's functional semantics and its very controlled parallelism in `with`-loops. Since variables in `sac` have write-once/read-only semantics, we do not have to take into account that they could change their value. Furthermore, parallelism only occurs in `with`-loops and while arbitrary variables can be read in the body `with`-loop, only a single variable is written to.

4 Implementation of our distributed memory backend

We added support for distributed memory architectures to the SAC compiler and runtime system based on a page-based software DSM system. Every node owns part of each distributed array and the owner computes principle applies. All accesses to remote data are performed through a local cache. To abstract from the physical network and provide portability, we utilise existing one-sided Remote Direct Memory Access (RDMA) communication libraries. Currently, we support GASNet [4], GPI-2 [13], ARMCI [16] and MPI-3. In order to add support for a communication library, one only has to provide implementations for a small set of operations. These include initialisation and shut down of the communication system, an operation to load a memory page from a remote node and barriers.

4.1 Distributed arrays and memory model

Distributed execution is triggered by `with`-loops that generate distributed arrays. The runtime system decides whether an array is distributed based on the size of the array, the number of compute nodes and the execution mode at allocation time (see Section 4.4 for execution modes). Arrays are always distributed blockwise along their first dimension. The minimum number of elements per node such that an array gets distributed can be configured at compile time.

In memory, a distributed array does not form one contiguous block, but instead it is split into *number-of-nodes* blocks of memory corresponding to the elements that are owned by each node. We will motivate the choice for this memory model in Section 4.5.

For an illustrative example of the memory model, see Figure 2. The example uses two arrays, denoted by different colours, with fourteen and eight elements, respectively, and four compute nodes. The numbers in the boxes denote the element indices. Every node owns a share of each distributed array. The portion of the array that is owned by a node is located in that node's shared segment (e.g. elements 0 - 3/0 - 1 of the first/second array on Node 0). Note that the array sizes were chosen to simplify the example; in practise only arrays that are some orders of magnitudes bigger would be distributed. Furthermore, we assume for this example that a memory page can hold three array elements only.

Each node's DSM memory consists of memory for the shared segment and memory for the local caches. At program startup, each node pins a configurable amout of memory for its shared segment and reserves an address space of the same size for the caches of each other node's data. (De-)allocation of distributed arrays in DSM memory is taken care of by an adapted version of the SAC private heap manager [11]. When a distributed array is allocated, the runtime system

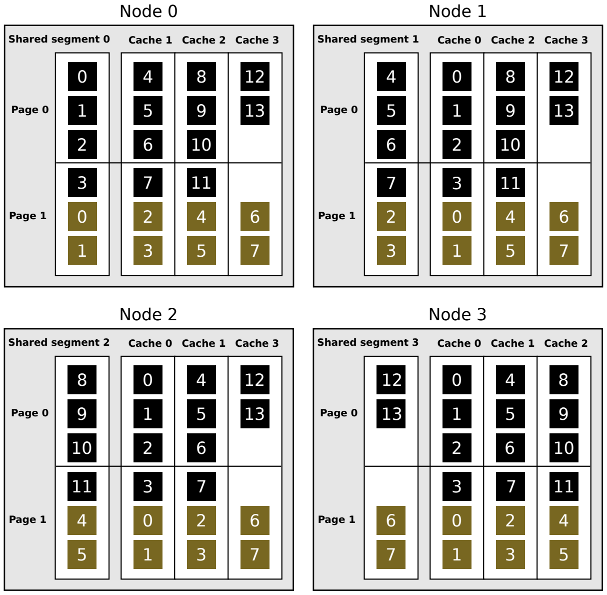


Fig. 2: Memory model for two distributed arrays (distinguished by different colours)

also reserves an address space of the same size within the local caches for all other nodes. To simplify locating array elements, the shared segment and caches are aligned. In the example, the second array starts at offset 4 in the shared DSM segment and all three cache segments on all four compute nodes. Non-distributed arrays, scalars and array descriptors are not allocated in DSM memory.

4.2 Array element pointer calculations

SAC supports multi-dimensional arrays; the translation of multi-dimensional array indices into vector offsets for memory accesses is taken care of by the compiler [3]. For the remainder of this paper, we assume that this conversion has already taken place. As explained in Section 4.1, a distributed array does not form a contiguous block of memory. The runtime system, therefore, needs to translate an offset to an array element to a pointer to the actual location of the element. This section describes how this is done and how we optimise this process.

In SAC, arrays have descriptors that hold a reference counter and, if not known at compile time, shape information. For each distributed array, we add two fields to the array descriptor: `first_elems` and `arr_offs`. The value of `first_elems` is the number of elements that are owned by each node except for the last node, which owns the remaining elements. The value of `arr_offs` is the offset at which the array starts within the shared segment of its owner node and within the cache for the owner at each other node. The formula for the pointer calculation is shown in Listing 1. The variable segments contains pointers to the local shared segment and the local caches; the rank of a node is the index of its segment within segments. The value of elem_offs is the offset of the requested element within the array assuming that the array would be allocated as one contiguous block of memory.

```
(segments[elem_offs / first_elems] + arr_offs) + (elem_offs % first_elems)
```

Listing 1: Formula for array element pointer calculations

In a naive implementation we would have to perform this pointer calculation for every access to an array element. However, we implemented three optimisations for write accesses, remote read accesses and local read accesses, respectively, so that the calculation can be avoided in most cases.

When writing distributed arrays we know that the elements we are writing to are local to the writing node because of the owner computes principle. We, therefore, simply keep a pointer to the start of the local portion of the array.

For remote read accesses we implemented a pointer cache. For each distributed array, we keep a pointer to the start of the array within the local cache for the node that owns the least recently accessed remote element of that array. In addition, we keep the offset of the first and last element that are owned by the same node.

For local read accesses we use the same pointer to the start of the local portion of the array that we use for write accesses. In addition, we keep the offset of the first and the last element that are local to the current node.

When a read access to an array element occurs, we first check whether the element is local to the current node by comparing its offset to the offsets of the first and last node that are local to the current node. If the element is local, we can use the pointer to the start of the local portion of the array in the local shared segment.

If the element is not local, we check whether it is owned by the same node as the last remote element of the same array that was accessed by comparing the offsets. If that is the case, we can use the pointer to the start of the array in the local cache for that node. Otherwise, we have to perform a pointer calculation as shown in Listing 1 to update the pointer cache.

4.3 Communication model and cache

According to the owner computes rule, a node only writes array elements that it owns. By contrast, every node can read all elements of a distributed array, including remote elements. This section describes the required communication for read accesses to remote array elements.

As mentioned in Section 4.1, the address space for the caches of remote elements is reserved when a distributed array is allocated. Initially, the caches are protected page-wise against all accesses by means of the `mprotect` system call. When a node tries to access remote data through its local cache, a `SIGSEGV` signal is raised. A custom handler then copies the appropriate memory page from the remote node into the local node's cache and allows accesses to it. Subsequent accesses to the same memory page can then be served directly from the local cache. The signal handler can calculate the requested array element and its location from the address where the segfault occurred. See Listing 1 for how to calculate the memory location of array elements.

When part of the cache becomes outdated, the corresponding memory pages are protected again. Distributed arrays are written in `with`-loops and we do not need any communication to trigger the required cache invalidations. Every node participates in the write operation and, therefore, knows that it has to invalidate the cache for that array on completion.

When a remote element is not in the local cache yet, we always load entire memory pages rather than single array elements. For an example, see Figure 2. When Node 0 first accesses Element 8 of the first array, Elements 9 and 10 will also be fetched from Node 2. Likewise, when Node 1 accesses Element 4 of the second array for the first time, Element 5 of the second and Element 11 of the first array will also be fetched from Node 2.

The rationale for loading entire pages is that thanks to advances in network technology, available bandwidth has increased so much that we can use it to hide latency [18]. Furthermore, the page-based approach allows us to use the operation system's memory page protection mechanism to decide whether an element is present in the cache or not.

4.4 Execution modes and barriers

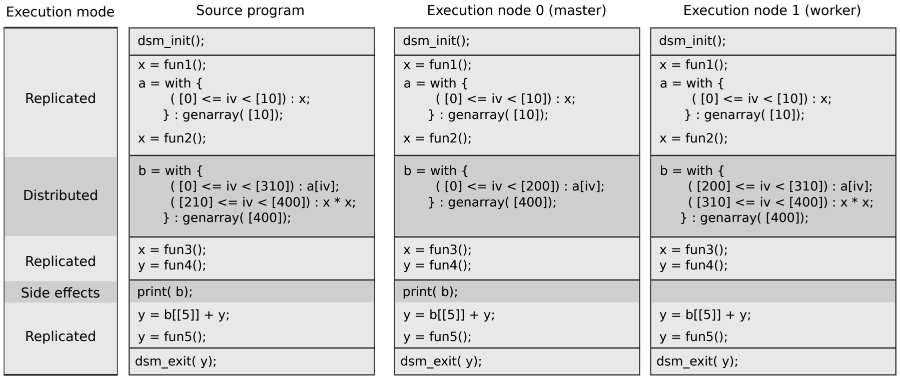
A distributed memory SAC program is always in one out of three execution modes: replicated, distributed or side effects execution mode. See Figure 3 for an illustrating example. In the following, we call the node with rank 0 master node and the remaining nodes worker nodes.

Program execution starts in replicated execution mode in which every node executes the same instructions on the same data. This way all nodes maintain the same execution environment without requiring communication.

In distributed execution mode, each node works on its share of the data. Currently, `genarray` and `modarray` `with`-loops are distributed iff the result array is distributed. Distributed memory SAC supports one level of distribution, an array and the `with`-loop that writes that array are not distributed if the program is already in distributed execution mode when the array is allocated.

In side effects execution mode, only the master node is executing and the workers are waiting until it is done. This is important because functions that have side effects, such as I/O, must not be executed more than once. If functions with side-effects yield any results, they are broadcast to the workers when the master is done.

In some cases we need barriers to preserve the correctness of the program in a distributed environment. For examples see Figure 3; the horizontal bars



Execution mode	Source program	Execution node 0 (master)	Execution node 1 (worker)
	dsm_init();	dsm_init();	dsm_init();

Replicated	x = fun1();
	a = with {
	([0] <= iv < [10]) : x;
	} : genarray([10]);
	x = fun2();
		x = fun1();
		a = with {
		([0] <= iv < [10]) : x;
		} : genarray([10]);
		x = fun2();
			x = fun1();
			a = with {
			([0] <= iv < [10]) : x;
			} : genarray([10]);
			x = fun2();

Distributed	b = with {
	([0] <= iv < [310]) : a[iv];
	([210] <= iv < [400]) : x * x;
	} : genarray([400]);
		b = with {
		([0] <= iv < [200]) : a[iv];
		} : genarray([400]);
			b = with {
			([200] <= iv < [310]) : a[iv];
			([310] <= iv < [400]) : x * x;
			} : genarray([400]);

Replicated	x = fun3();
	y = fun4();
		x = fun3();
		y = fun4();
			x = fun3();
			y = fun4();

Side effects	print(b);	print(b);	

Replicated	y = b[[5]] + y;
	y = fun5();
		y = b[[5]] + y;
		y = fun5();
			y = b[[5]] + y;
			y = fun5();

	dsm_exit(y);	dsm_exit(y);	dsm_exit(y);

Fig. 3: Execution modes and barriers (horizontal bars)

denote barriers. In general, we require barriers after program startup and before program termination, before and after a distributed `with`-loop and before a function application with side effects.

The barrier after a distributed `with`-loop ensures that no stale data is read by other nodes because there were write accesses to the distributed array in the `with`-loop. The barrier before a distributed `with`-loop ensures that, in case memory is reused (see [12] for SAC's memory management), no other node needs to read the old data anymore before it is overwritten.

4.5 Motivation for memory model

As described in Section 4.1, distributed arrays do not form one contiguous block, but instead are split into *number-of-nodes* blocks of memory corresponding to the elements that are owned by each node. We explained in Section 4.2 that it is relatively expensive to calculate pointers to array elements with this memory model and proposed a pointer cache as a solution. Given this disadvantage, why do we propose the described memory model? For our argumentation we will assume that we use a *page-based* DSM system. We will, therefore, first elaborate on the reasons why we decided to build a *page-based* DSM system: to hide latency and to avoid overheads when checking whether an element is present in the cache.

On a cache miss, we fetch a whole memory page rather than a single element from the remote node that owns the element. Subsequent accesses to neighbouring elements can then be served from the cache. This allows us to use the available bandwidth to hide latency. In addition, if we fetch whole memory pages, we can use the operating system's page protection mechanism to decide whether a page is present in the cache or not. If a page is not present in the cache, a `SIGSEGV` signal is raised when we try to access it and the fetch from the remote node is taken care of by our custom signal handler. If a page is present in the cache, however, the access simply returns the data. The alternative to using the page protection mechanism would be to keep track of the cached elements ourselves, but that would involve a search in a possibly large data structure. This search would incur additional overheads, also in the case that an element is already present in the cache.

Having decided that we want to use a page based DSM system, why do we use the described memory model? SAC supports multi-dimensional arrays and `with`-loops that generate multi-dimensional arrays are compiled to complex nested loop structures with a loop for each array dimension. We need to make sure that the distribution happens along a single dimension; in practise along the outermost dimension. Otherwise, the iteration of the index space becomes impractically complex, especially when considering that the size and dimensionality of arrays is often not known at compile time.

We have established that we want to use a page-based DSM system and that the distribution of the array should happen along the outermost dimension. If an array was to form a single contiguous form of memory we would then have to partition it at memory page borders. However, we have also established that the distribution should happen along the outermost dimension. Unfortunately, these two demands generally cannot be met at the same time.

Another benefit of our memory model is that it allows us to solve larger problems. With contiguous arrays, we would need to allocate the entire array within the DSM segment so that remote nodes can read the local portion of it. Unfortunately, the size of the DSM segment is limited by hardware constraints. In any case, it cannot be larger than the node's physical memory. By contrast, in our memory model, we only allocate the local part of the array within the DSM segment. The caches are allocated outside of the DSM segment using `mmap`. Until a memory page is accessed for the first time, only an address space is reserved but no phyiscal memory is provided.

5 Evaluation of our distributed memory backend

We evaluate the performance of our distributed memory backend for SAC by means of experiments in the areas of image convolution, matrix multiplication and N-body simulation. In the following, we will first describe the experimental setup and then discuss the results of the individual experiments.

5.1 Experimental setup

All experiments were performed on the VU cluster side of the DAS-4 supercomputer system [2]. The VU cluster side consists of 74 dual quad-core 2.4 GHz compute nodes with 24 GB of memory each. The nodes are interconnected by Gigabit Ethernet as well as high speed InfiniBand. We used the following versions of the supported communication libraries for our experiments: GASNet 1.24.0, GPI-2 1.1.1, ARMCI as included in Global Arrays 5.3 and the Open MPI 1.6.5 implementation of MPI-3.

In our experiments, we compare the runtimes of the program compiled for our distributed memory backend (`dm`) to the runtimes of the sequential SAC program (`seq`). With the distributed program, we start each process on a separate compute node. For $N \leq 8$ (as the nodes of the DAS-4 system have eight cores), we also compare the performance of our distributed memory backend program run by multiple processes on a single node (`dm-sn`) to the performance of the multi-threaded SAC program `mt`.

For all included measurements, we compared the output of the distributed memory backend program to the output of the sequential program to ensure that the program yields correct results. We measure the kernel execution time of the calculations and not the total execution time of the program. The reason is that the setup of the communication libraries and the printing of the result arrays to check the correctness take a considerable amount of time and that would otherwise distort our results. For real-world applications, the compute time would be much longer, whereas the setup time remains nearly constant and, thus, can be neglected.

We performed all experiments at least three times or more often if there was a high variance in the results. From all measurements, we take the minimum execution time for each program version rather than the average execution time. Our justification is that there may be background processes running on the compute nodes that have an influence on our experiments. All reported speedups are with respect to the sequential SAC program (`seq`).

5.2 Image convolution

First, we present our image convolution experiments. We include image convolution in our evaluation because it is a simple application where array element accesses show a high degree of locality. We have optimised our implementation for that by fetching entire pages on a cache miss and by using optimisations such as array pointer caches (see Section 4.2).

The `gaussBlurOpt` test program performs twenty iterations of a 3 x 3 kernel Gaussian blur on a 50,000 x 8,000 = 400,000,000 elements integer array. Figure 4 shows the performance results for `gaussBlurOpt`. For this program, we achieve speedups of more than 80% of linear for up to sixteen nodes.

5.3 Matrix multiplication

We also include experiments with matrix multiplication, because, compared to image convolution, it requires more communication. In this way, the matrix multiplication experiments are a stress test for the communication performance of our distributed memory backend for SAC.

The `matmulBigDiff` program performs ten iterations of a multiplication of two matrices with 2,000 x 2,000 = 4,000,000 double-precision floating point elements each. Implementation-wise, we first transpose the second matrix before we calculate the result matrix. Figure 5 shows our measurements for the

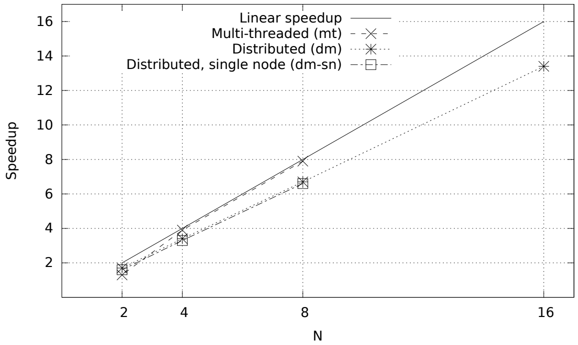


Fig. 4: Speedups for the `gaussBlurOpt` program (twenty iterations of a 3 x 3 kernel Gaussian blur on a 50,000 x 8,000 = 400,000,000 elements integer array)

`matmulBigDiff` program: for eight nodes we achieve a speedup of 3.2 (40% of linear) and for sixteen nodes a speedup of 4.2 (26% of linear).

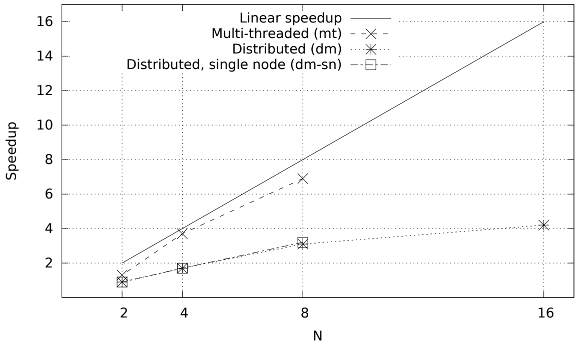


Fig. 5: Speedups for the `matmulBigDiff` program (ten iterations of a multiplication of two matrices with 2,000 x 2,000 = 4,000,000 double-precision floating point elements each)

5.4 N-Body simulation

Finally, we present the measurements for our all-pairs N-body problem experiments. The SICSA N-body challenge simulates the movements of a system of planets in three-dimensional space over time. Our program is based on the SAC implementation proposed in [20].

The nbodyBig program performs fifty iterations for 16,384 planets. Figure 6 show the measurements for the nbodyBig program. We achieve approximately 50% of linear speedups for up to sixteen nodes.

For the nbodyBig program, we also compare the minimum runtimes with the different communication libraries GASNet, ARMCI, GPI-2 and MPI-3. In Figure 7, we can see that MPI shows the weakest overall performance. Overall, GASNet is slightly faster than ARMCI and GPI-2.

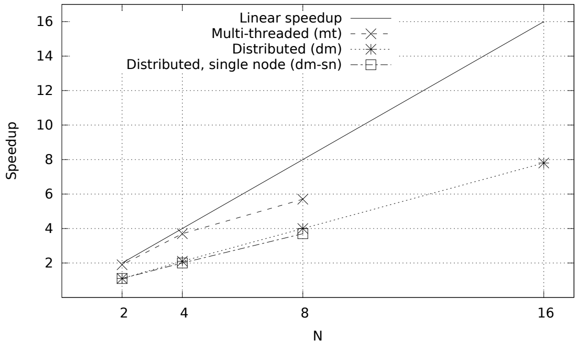


Fig. 6: Speedups for the nbodyBig program (N-body simulation: movements of 16,384 planets, 50 iterations)

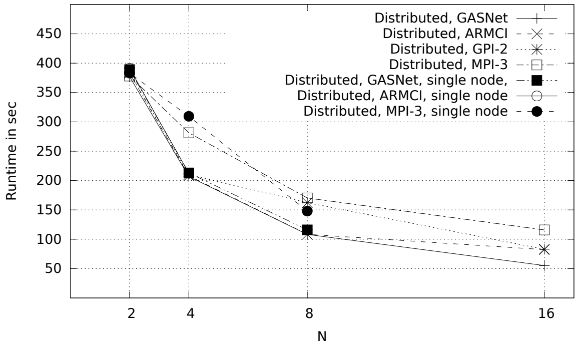


Fig. 7: Minimum runtimes with different communication libraries for the nbodyBig program (N-body simulation: movements of 16,384 planets, 50 iterations)

6 Conclusions and future work

6.1 Conclusion

In this paper, we have presented our implementation of a new compiler backend for SAC that supports symmetric distributed memory architectures like clusters of workstations. A particular challenge in doing so is upholding SAC's promise of entirely compiler-directed exploitation of concurrency.

We propose a DSM-based implementation where all accesses to remote data go through large local caches. Initially, the caches are protected by means of the `mprotect` system call. When a memory page is first accessed, a `SIGSEGV` signal is raised. A custom signal handler fetches the requested data from the remote node and subsequent accesses to the same data can be served directly from the cache.

While there is a lot of work to be done, our first results are promising. For our convolution experiments, we achieve 80% of linear speedups, for our N-body simulation approximately 50% of linear speedups and for matrix multiplication about one third of linear speedups.

6.2 Future Work

Possible future research directions lie in the areas of general performance improvements, the combination with multi-threading, cache eviction and distributed I/O. In the following, we briefly elaborate on these topics.

We want to improve overall performance by reducing the number of barriers. Furthermore, we want to make read operations to distributed arrays more efficient by avoiding locality checks and/or reducing overheads caused by them.

To fully utilise clusters of multi-core compute nodes, we want to combine the distributed memory backend with SAC's multi-threaded execution facilities [8]. We expect that we can achieve higher speedups with a hybrid solution that combines distributed execution and multi-threading.

Other than speeding up program execution, distributed execution has another advantage: It allows us to solve problems that do not fit into the memory of a single node. This is already possible to some extent in our solution, but to support the general case, we would need to add a cache eviction scheme.

Currently, functions that have side effects including I/O are only executed by the master node. We decided for this implementation to ensure that existing SAC libraries work correctly with the distributed memory backend. However, in many situations it would be more efficient to distribute I/O operations.

References

1. TOP500 supercomputer sites (2014), `http://top500.org/`, accessed on 19 February 2015
2. DAS-4: Distributed ASCI supercomputer 4 (2015), `http://www.cs.vu.nl/das4/home.shtml`, accessed on 25 July 2015

3. Bernecky, R., Herhut, S., Scholz, S.B., Trojahner, K., Grelck, C., Shafarenko, A.: Index Vector Elimination - Making Index Vectors Affordable, pp. 19–36. Implementation and Application of Functional Languages, Springer (2007)
4. Bonachea, D.: GASNet specification, v1. 1. Tech. rep., University of California at Berkeley (2002)
5. Chamberlain, B.L., Callahan, D., Zima, H.P.: Parallel programmability and the Chapel language. International Journal of High Performance Computing Applications 21(3), 291–312 (2007)
6. Diogo, M., Grelck, C.: Towards heterogeneous computing without heterogeneous programming, pp. 279–294. Trends in Functional Programming, Springer (2013)
7. Dongarra, J.J., der Steen, A.V.: High-performance computing systems: Status and outlook. Acta Numerica 21, 379–474 (2012)
8. Grelck, C.: A multithreaded compiler backend for high-level array programming. In: Applied Informatics. pp. 478–484 (2003)
9. Grelck, C.: Single Assignment C (SAC): High Productivity Meets High Performance, pp. 207–278. Central European Functional Programming School, Springer (2012)
10. Grelck, C., Scholz, S.B.: SAC: off-the-shelf support for data-parallelism on multicores. In: Proceedings of the 2007 workshop on Declarative aspects of multicore programming. pp. 25–33. ACM (2007)
11. Grelck, C., Scholz, S.B.: Efficient heap management for declarative data parallel programming on multicores. In: 3rd Workshop on Declarative Aspects of Multicore Programming (DAMP 2008), San Francisco, CA, USA. pp. 17–31 (2008)
12. Grelck, C., Trojahner, K.: Implicit memory management for SAC. In: Implementation and Application of Functional Languages, 16th International Workshop, IFL. vol. 4, pp. 335–348 (2004)
13. Grünewald, D., Simmendinger, C.: The GASPI API specification and its implementation GPI 2.0. In: 7th International Conference on PGAS Programming Models. vol. 243 (2013)
14. Guo, J., Thiyagalingam, J., Scholz, S.B.: Breaking the GPU programming barrier with the auto-parallelising SAC compiler. In: Proceedings of the sixth workshop on Declarative aspects of multicore programming. pp. 15–24. ACM (2011)
15. Herhut, S., Joslin, C., Scholz, S.B., Grelck, C.: Truly nested data-parallelism: compiling SAC for the Microgrid architecture. Draft proceedings of the 21st International Symposium on Implementation and Application of Functional Languages (2009)
16. Nieplocha, J., Carpenter, B.: ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems, pp. 533–546. Parallel and Distributed Processing, Springer (1999)
17. Nitzberg, B., Lo, V.: Distributed shared memory: A survey of issues and algorithms. Distributed Shared Memory-Concepts and Systems pp. 42–50 (1991)
18. Ramesh, B., Ribbens, C.J., Varadarajan, S.: Is it time to rethink distributed shared memory systems? In: Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on. pp. 212–219 (2011), iD: 1
19. Ramesh, B.: Samhita: Virtual shared memory for non-cache-coherent systems (2013)
20. Šinkarovs, A., Scholz, S.B., Bernecky, R., Douma, R., Grelck, C.: SAC/C formulations of the all-pairs N-body problem and their performance on SMPs and GPGPUs. Concurrency and Computation: Practice and Experience 26(4), 952–971 (2014)

Tree-Like Grammars and Separation Logic
(Extended Abstract)

Christoph Matheja, Christina Jansen, and Thomas Noll

Software Modeling and Verification Group
RWTH Aachen University, Germany
http://moves.rwth-aachen.de/

Abstract. Separation Logic with inductive predicate definitions (SL) and hyperedge replacement grammars (HRG) are established formalisms to describe the abstract shape of data structures maintained by heap-manipulating programs. Fragments of both formalisms are known to coincide, and neither the entailment problem for SL nor its counterpart for HRGs, the inclusion problem, are decidable in general.
We introduce *tree-like grammars* (TLG), a fragment of HRGs with a decidable inclusion problem. By the correspondence between HRGs and SL, we simultaneously obtain an equivalent SL fragment ($\mathtt{SL}_{\mathtt{tl}}$) featuring some remarkable properties including a decidable entailment problem.

1 Introduction

Symbolic execution of heap-manipulating programs builds upon abstractions to obtain finite descriptions of dynamic data structures, like linked lists and trees. Proposed abstraction approaches employ, amongst others, Separation Logic with inductive predicate definitions (SL) [21, 2, 16] and hyperedge replacement grammars (HRG) [12, 15].

While these formalisms are intuitive and expressive, important problems are undecidable. In particular, the *entailment problem* for SL [5, 1], i.e. the question whether all models of a formula φ are also models of another formula ψ, as well as its graph-theoretical counterpart, the *inclusion problem* for HRGs [12], are undecidable in general. Unfortunately, as stated by Brotherston, Distefano and Peterson [4], "effective procedures for establishing entailments are at the foundation of automatic verification based on Separation Logic". Consequently, SL-based verification tools, such as SLayer [3] and Predator [13], often restrict themselves to the analysis of list-like data structures, where the entailment problem is known to be decidable [2]. VeriFast [17], Hip/Sleek [7] and Cyclist [4] allow general user-specified predicates, but are incomplete and/or require additional user interaction. The largest known fragment of SL featuring both inductive predicate definitions and a decidable entailment problem is Separation Logic with bounded tree width ($\mathtt{SL}_{\mathtt{btw}}$) [16].

Approaches based on graph grammars suffer from the undecidability of the related inclusion problem: Lee et al. [20] propose the use of graph grammars for shape analysis, but their approach is restricted to trees. JUGGRNAUT [15] allows the user to specify the shape of dynamic data structures by an HRG, but relies on an approximation to check whether newly computed abstractions are subsumed by previously encountered ones. Hence, finding more general fragments of SL and HRGs with good decidability properties is highly desirable.

This paper investigates fragments of HRGs with a decidable inclusion problem. In a nutshell, HRGs are a natural extension of context-free word grammars specifying the replacement of nonterminal-labelled edges by graphs (cf. [14]). Common notions and results for context-free word languages, e.g. decidability of the emptiness problem and existence of derivation trees, can be lifted to HRGs (cf. [22]) which justifies the alternative name "context-free graph grammars".

Most of our results stand on two pillars. To introduce these two pillars as well as to summarise our main results here, the utilisation of some formal notation and concepts is indispensable. Corresponding definitions and detailed explanations can be found in the successive sections. The first pillar is an extension of the well-known fact that context-free word languages are closed under intersection with regular word languages, which are, by Büchi's famous theorem [6], exactly the word languages definable in monadic second-order logic (MSO).

Lemma 1 (Courcelle [8]) *For each HRG G and $\mathtt{MSO}_2$ sentence φ, one can construct an HRG G' such that $L(G') = L(G) \cap L(\varphi) = \{H \in L(G) \mid \underline{H} \models \varphi\}$.*

Here, $\mathtt{MSO}_2$ means MSO over graphs with quantification over nodes and edges.

The second pillar is the close connection between a fragment of HRGs - called data structure grammars (DSG) - and a fragment of SL studied by Dodds [11] and Jansen et al. [18].

Lemma 2 (Jansen et al. [18]) *Every SL formula can be translated into a language-equivalent data structure grammar and vice versa.*

The overall goal of this paper is to develop fragments of HRGs which can be translated into $\mathtt{MSO}_2$. Then it directly follows from Lemma 1 that the resulting classes of languages have a decidable inclusion problem and are closed under union, intersection and difference as well as under intersection with general context-free graph languages. By Lemma 2, we obtain analogous results for equivalent SL fragments.

The largest fragment we propose are *tree-like grammars* (TLG). Intuitively, every graph H generated by a TLG allows to reconstruct one of its derivation trees by identifying certain nodes, the *anchor* nodes, with positions in a derivation tree. Furthermore, each edge of H is uniquely associated with one of these anchor nodes. These properties allow for each graph H generated by a given TLG G to first encode a derivation tree t in $\mathtt{MSO}_2$ and then to verify that H is in fact the graph derived by G according to t. Our main result is that the two informally stated properties from above guarantee $\mathtt{MSO}_2$-definability.

Theorem 1. *For each TLG G, there exists an $\mathtt{MSO}_2$ sentence φ_G such that for each $H \in \mathtt{HG}$, $H \in L(G)$ if and only if $\underline{H} \models \varphi_G$.*

TLGs are introduced in detail in Section 4. Furthermore, we study the fragment of *tree-like Separation Logic* ($\texttt{SL}_{\texttt{tl}}$, cf. Section 5) which is equivalent to TLGs generating heaps rather than arbitrary graphs.

Definition 1. *An $\texttt{SL}_{tl}$ environment is an $\texttt{SL}$ environment Γ where every disjunct $\varphi(x_1, ..., x_n)$ of every predicate definition meets the following conditions:*

- Anchoredness*: All pointer assertions $y.s \mapsto z$ occurring in φ contain the first parameter x_1 of φ, either on their left-hand or right-hand side, i.e. $x_1 = y$ or $x_1 = z$.*
- Connectedness*: The first parameter of every predicate call in φ occurs in some pointer assertion of φ.*
- Distinctness*: x_1 is unequal to the first parameter of every predicate call occurring in Γ.*

By Lemma 2, our results on TLGs also hold for $\texttt{SL}_{\texttt{tl}}$. Thus, $\texttt{SL}_{\texttt{tl}}$ has the following remarkable properties:

1. The satisfiability as well as the *extended* entailment problem, i.e. the question whether an *arbitrary* $\texttt{SL}$ formula φ entails an $\texttt{SL}_{\texttt{tl}}$ formula ψ, are decidable.
2. Although negation and conjunction are restricted to pure formulae, $\texttt{SL}_{\texttt{tl}}$ is closed under intersection and difference.

Regarding expressiveness, common data structures like (cyclic) lists, trees, in-trees, $n \times k$-grids for fixed k and combinations thereof are $\texttt{SL}_{\texttt{tl}}$-definable. In particular, we show that $\texttt{SL}_{\texttt{tl}}$ is strictly more expressive than $\texttt{SL}_{\texttt{btw}}$. The same holds for an entirely syntactic fragment of TLGs, called Δ-DSGs, and a corresponding fragment of $\texttt{SL}_{\texttt{tl}}$.

A full version of this paper containing further details and proofs has recently been submitted to APLAS[1].

The remainder of this paper is structured as follows. Section 2 very briefly recapitulates standard definitions on $\texttt{SL}$ and $\texttt{MSO}$, while Section 3 covers essential concepts of hypergraphs and HRGs. The fragment of TLGs and its $\texttt{MSO}$-definability result is introduced in Section 4. Our results on TLGs are transferred to $\texttt{SL}$ and discussed in Section 5. Finally, Section 6 concludes.

2 Preliminaries

This section introduces our notation and briefly recapitulates trees, graphs, $\texttt{MSO}_2$, and $\texttt{SL}$. On first reading, the well-informed reader might want to skip this part.

[1] http://pl.postech.ac.kr/aplas2015/

Notation Given a set S, $S^{\star}$ denotes all finite sequences over S. For $s, s' \in S^{\star}$, $s.s'$ denotes their concatenation, the i-th element of s is denoted by $s(i)$ and the set of all of its elements is denoted by $\lfloor s \rfloor$. A ranked alphabet is a finite set S with ranking function $rk_S : S \to \mathbb{N}$ and maximal rank $\Re(S)$. We write $\{x_1 \mapsto y_1, \ldots, x_m \mapsto y_m\}$ to denote a finite (partial) function f with domain $\mathrm{dom}(f) = \{x_1, \ldots, x_m\}$ and co-domain $\{y_1, \ldots, y_m\}$ such that $f(x_i) = y_i$ for each $i \in [m] = [1, m] = \{1, 2, \ldots, m\}$. The operators $\uplus$ and $\uplus$ denote the disjoint union of two sets and two functions, respectively.

Trees Given a ranked alphabet S, a *tree* over S is a finite function $t : \mathrm{dom}(t) \to S$ such that $\emptyset \neq \mathrm{dom}(t) \subseteq \mathbb{N}^{\star}$, $\mathrm{dom}(t)$ is prefix closed and for all $x \in \mathrm{dom}(t)$, $\{i \in \mathbb{N} \mid x.i \in \mathrm{dom}(t)\} = [rk_S(t(x))]$. $x \in \mathrm{dom}(t)$ is a (proper) prefix of $y \in \mathrm{dom}(t)$, written $x \prec y$, if $y = x.i.z$ for some $i \in \mathbb{N}$ and $z \in \mathbb{N}^{\star}$. The subtree of t with root $x \in \mathrm{dom}(t)$ is given by $t|_x : \{y \mid x.y \in \mathrm{dom}(t)\} \to S : y \mapsto t(x.y)$.

Graphs An edge-labelled graph over an alphabet S is a tuple $H = (V, E)$ with a finite set of nodes V and edge relation $E \subseteq V \times S \times V$. With each graph H we associate the relational structure $\underline{H} = (V \uplus E, \mathrm{src}, \mathrm{tgt}, (E_s)_{s \in S})$ where src and tgt are the binary source and target relations given by $\mathrm{src} := \{(u, e) \mid e = (u, s, v) \in E\}$, $\mathrm{tgt} := \{(e, v) \mid e = (u, s, v) \in E\}$. For each $s \in S$, there is a unary relation $E_s := \{(u, s, v) \in E \mid u, v \in V\}$ collecting all edges labelled with s.

Monadic Second-Order Logic over Graphs Given a finite alphabet S, the syntax of $\texttt{MSO}_2$ is given by:

$$\varphi ::= E_s(x) \mid \mathrm{src}(x, y) \mid \mathrm{tgt}(x, y) \mid X(x) \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi \mid \exists x : \varphi \mid \exists X : \varphi \mid x = y$$

where x, y are first-order variables, X is a second-order variable and $s \in S$. For a graph $H = (V, E)$, we write $\underline{H}, \jmath \models \varphi$ iff $\underline{H}$ satisfies φ where $\jmath$ is an interpretation mapping every free first-order variable to an element of $V \uplus E$ and every second-order variable to a subset of either V or E, respectively. The semantics of $\models$ is standard (cf. [10]). Note that the semantics of $\mathrm{src}, \mathrm{tgt}$ and E_s has been given explicitly in the definition of $\underline{H}$.

Heaps Similarly to the typical RAM model, a heap is understood as a set of locations *Loc*, whose values are interpreted as pointers to other locations. Formally, we assume $Loc := \mathbb{N}$ and define a *heap* as a partial mapping $h : Loc \to Loc \uplus \{\mathbf{null}\}$. The set of all heaps is denoted by $\texttt{He}$. Let Σ be a finite alphabet of *selectors* equipped with an injective ordering function $\mathrm{cn} : \Sigma \to [0, |\Sigma| - 1]$. We assume a heap to consist of objects equipped with finitely many pointer variables which are modelled by reserving exactly $|\Sigma|$ successive locations. Hence, for a heap containing n objects, $dom(h) = [n \cdot |\Sigma|]$.

Separation Logic with Recursive Definitions We consider a fragment of Separation Logic, similar to Separation Logic with recursive definitions in [16, 18], in which negation ¬, `true`, and conjunction ∧ in spatial formulae are disallowed. Let *Pred* be a set of predicate names. The *syntax of* `SL` is given by:

$$
\begin{array}{ll}
E ::= x \mid \mathbf{null} & \\
P ::= x = y \mid P \wedge P & \text{pure formulae} \\
F ::= \mathbf{emp} \mid x.s \mapsto E \mid F * F \mid \exists x : F \mid \sigma(x_1, ..., x_n) & \text{spatial formulae} \\
S ::= F \mid S \vee S \mid S \wedge P & \texttt{SL}\text{ formulae}
\end{array}
$$

where $x, y, x_1, ..., x_n \in \mathit{Var}$, $s \in \Sigma$ and $\sigma \in \mathit{Pred}$. The formula $x.s \mapsto E$ is called a *pointer assertion*, $\sigma(x_1, ..., x_n)$ a *predicate call*.

Note that we do not require that all selectors of a given variable are defined by a single pointer assertion, i.e. we are less strict about pointer assertions than other fragments proposed in the literature, e.g. in [16]. Furthermore, it is straightforward to add program variables to `SL`, which we omitted for the sake of simplicity. To improve readability, we write $x.(s_1, \ldots, s_k) \mapsto (y_1, \ldots, y_k)$ as a shortcut for $x.s_1 \mapsto y_1 * \ldots * x.s_k \mapsto y_k$.

Predicate calls are interpreted by means of predicate definitions. A *predicate definition* for $\sigma \in \mathit{Pred}$ is of the form $\sigma(x_1, ..., x_n) := \sigma_1 \vee ... \vee \sigma_m$ where $m, n \in \mathbb{N}$, σ_j is a formula of the form $F \wedge P$, and $x_1, ..., x_n \in \mathit{Var}$ are pairwise distinct and exactly the free variables of σ_j for each $j \in [m]$. The disjunction $\sigma_1 \vee ... \vee \sigma_m$ is called the *body* of the predicate. An *environment* is a set of predicate definitions. *Env* denotes the set of all environments.

The semantics of a predicate call $\sigma(x_1, ..., x_n)$, $\sigma \in \mathit{Pred}$, w.r.t. an environment $\Gamma \in \mathit{Env}$ is given by the predicate interpretation η_Γ. It is defined as the least set of location sequences instantiating the arguments $x_1, \ldots, x_n$ and heaps that fulfil the unrolling of the predicate body. We refer to [18] for a formal definition.

The semantics of the remaining `SL` constructs is determined by the standard semantics of first-order logic and the following, where $\jmath$ is an interpretation of variables as introduced for `MSO`$_2$:

$$
\begin{array}{lll}
h, \jmath, \eta_\Gamma \models x.s \mapsto \mathbf{null} & \Leftrightarrow & \mathrm{dom}(h) = \{\jmath(x) + cn(s)\}, h(\jmath(x) + cn(s)) = \mathbf{null} \\
h, \jmath, \eta_\Gamma \models x.s \mapsto y & \Leftrightarrow & \mathrm{dom}(h) = \{\jmath(x) + cn(s)\}, h(\jmath(x) + cn(s)) = \jmath(y) \\
h, \jmath, \eta_\Gamma \models \sigma(x_1, ..., x_n) & \Leftrightarrow & ((\jmath(x_1), ..., \jmath(x_n)), h) \in \eta_\Gamma(\sigma) \\
h, \jmath, \eta_\Gamma \models \varphi_1 * \varphi_2 & \Leftrightarrow & \exists h_1, h_2 : h = h_1 \uplus h_2, h_1, \jmath, \eta_\Gamma \models \varphi_1,\ h_2, \jmath, \eta_\Gamma \models \varphi_2
\end{array}
$$

A variable $x \in \mathit{Var}$ is said to be *allocated* in a formula if it (or a variable y with $y = x$) occurs on the left-hand side of a pointer assertion.

From now on, we assume that all existentially quantified variables are eventually allocated. This requirement is similar to the "establishment" condition in [16]. With this assumption, the inequality operator for logical variables $x \neq y$ is redundant with respect to the expressive power of the formalism, because $x.s \mapsto z * y.s \mapsto z$ already implies that $\jmath(x) \neq \jmath(y)$ in all heaps satisfying the formula. Thus, we assume that two existentially quantified variables refer to different locations if not stated otherwise by a pure formula.

3 Context-Free Graph Grammars

This section introduces HRGs together with some of their properties relevant for the remainder of this paper. For a comprehensive introduction, we refer to [14, 22].

Let $\Sigma_N := \Sigma \uplus N$ be a ranked alphabet consisting of terminal symbols Σ and nonterminal symbols N.

Definition 2 (Hypergraph). *A* labelled hypergraph *(HG) over Σ_N is a tuple $H = (V, E, \mathrm{att}, \mathrm{lab}, \mathrm{ext})$ where V and E are disjoint sets of nodes and hyperedges,* $\mathrm{att} : E \to V^{\star}$ *maps each hyperedge to a sequence of attached nodes such that* $|\mathrm{att}(e)| = \mathit{rk}_{\Sigma_N}(\mathrm{lab}(e))$*,* $\mathrm{lab} : E \to \Sigma_N$ *is a labelling function, and* $\mathrm{ext} \in V^{\star}$ *a sequence of external nodes. The set of all HGs over Σ_N is denoted by $\mathtt{HG}_{\Sigma_N}$.*

Note that we allow attachments of hyperedges as well as the sequence of external nodes to contain repetitions. Hyperedges with a label from Σ are called *terminal edges*, *nonterminal* otherwise. The set of terminal (nonterminal) hyperedges of an HG H is denoted by E_H^{Σ} (E_H^{N}, respectively). In this paper, we assume $\mathit{rk}_{\Sigma_N}(s) = 2$ for each $s \in \Sigma$. Moreover, a hyperedge e with $\mathrm{lab}(e) = s \in \Sigma$ and $\mathrm{att}(e) = u.v$ is interpreted as a directed edge from u to v. The relational structure corresponding to $H \in \mathtt{HG}_{\Sigma}$ is $\underline{H} := \underline{[H]}$, where the (conventional) graph $[H]$ is defined as $[H] = (V_H, E)$, $E := \{(\mathrm{att}_H(e)(1), \mathrm{lab}_H(e), \mathrm{att}_H(e)(2)) \mid e \in E_H\}$.

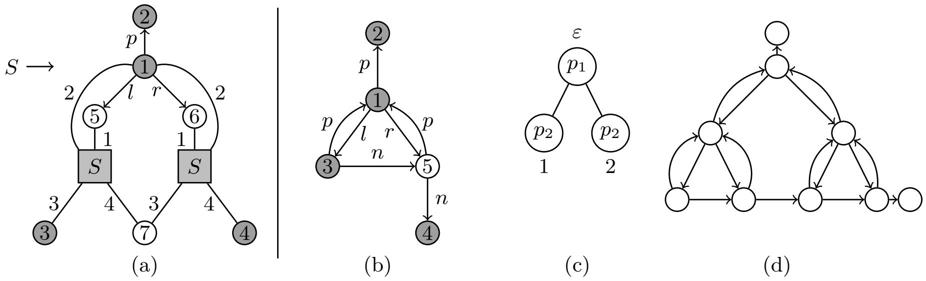
Example 1. As an example, consider the HG illustrated in Figure 1(a). For referencing purpose, we provide a unique index $i \in [|V|]$ inside of each node u_i represented by a circle. External nodes are shaded. For simplicity, we assume them to be ordered according to the provided index. Terminal edges are drawn as directed, labelled edges and nonterminal edges as square boxes with their label inside. The ordinals pictured next to the connections of a nonterminal hyperedge denote the position of the attached nodes in the attachment sequence. For example, if e is the leftmost nonterminal hyperedge in Figure 1(a), $\mathrm{att}(e) = u_5.u_1.u_3.u_7$.

Two HGs H, H' are isomorphic, written $H \cong H'$, if they are identical up to renaming of nodes and edges. In this paper, we will not distinguish between isomorphic HGs. The disjoint union of $H, H' \in \mathtt{HG}_{\Sigma_N}$ is denoted by $H \uplus H'$.

The main concept to specify (infinite) sets of HGs in terms of context-free graph grammars is the replacement of a nonterminal hyperedge by a finite HG. Intuitively, a nonterminal hyperedge e is replaced by an HG H by first removing e, inserting a disjoint copy of H and identifying the nodes originally attached to e with the sequence of external nodes of H. This is formally expressed by a quotient.

Definition 3 (Hypergraph Quotient). *Let $H \in \mathtt{HG}_{\Sigma_N}$, $R \subseteq V_H \times V_H$ be an equivalence relation and $[u]_{/R} = \{v \in V_H \mid (u, v) \in R\}$ the equivalence class of $u \in V_H$, which is canonically extended to sequences of nodes. The R-quotient graph of H is $[H]_{/R} = (V, E, \mathrm{att}, \mathrm{lab}, \mathrm{ext})$, where $V = \{[u]_{/R} \mid u \in V_H\}$, $E = E_H$,* $\mathrm{att} = \{e \mapsto [\mathrm{att}_H(e)]_{/R} \mid e \in E_H\}$*,* $\mathrm{lab} = \mathrm{lab}_H$*,* $\mathrm{ext} = [\mathrm{ext}_H]_{/R}$*.*

Definition 4 (Hyperedge Replacement). *Let $H, K \in \mathtt{HG}_{\Sigma_N}$ be hypergraphs with disjoint nodes and hyperedges, $e \in E_H^N$ with $rk_{\Sigma_N}(e) = k = |\mathrm{ext}_K|$. Let $V = V_H \uplus V_K$, and ${}_{H,e}{\approx}_K \subseteq V \times V$ be the least equivalence relation containing $\{(\mathrm{att}_H(e)(i), \mathrm{ext}_K(i)) \mid i \in [k]\}$. Then the HG obtained from replacing e by K is $H[e/K] := [(H \setminus \{e\} \uplus K)]_{/\ {}_{H,e}{\approx}_K}$ where $H \setminus \{e\}$ is the HG H in which e has been removed. Moreover, two nodes $u, v \in V$ are* merged *by $H[e/K]$ if $u \neq v$ and $u\ {}_{H,e}{\approx}_K\ v$.*



production rule p_1 production rule p_2 derivation tree t *yield*(t)

Fig. 1. HRG *TLL* with two production rules p_1 and p_2

We now formally introduce context-free graph grammars based on hyperedge replacement.

Definition 5 (Hyperedge Replacement Grammar). *An HRG is a 3-tuple $G = (\Sigma_N, P, S)$ where Σ_N is a ranked alphabet, $S \in N$ is the initial symbol and $P \subseteq N \times \mathtt{HG}_{\Sigma_N}$ is a finite set of production rules such that $rk_{\Sigma_N}(X) = |\mathrm{ext}_H| > 0$ for each $(X, H) \in P$. The class of all HRGs is denoted by $\mathtt{HRG}$.*

Given $p = (X, H) \in P$, we write $\mathrm{lhs}(p)$ and $\mathrm{rhs}(p)$ to denote X and H, respectively. To improve readability, we write p instead of $\mathrm{lhs}(p)$ or $\mathrm{rhs}(p)$ whenever the context is clear.

Example 2. The HRG *TLL* depicted in Figure 1(a),(b) will serve as a running example. It consists of one nonterminal symbol S, four terminal symbols l, r, p, n and two production rules p_1, p_2.

A key feature of HRGs is that the order in which nonterminal hyperedges are replaced is irrelevant, i.e. HRGs are confluent (cf. [14, 22]). Thus, derivations of HRGs can be described by derivation trees. Towards a formal definition, we assume that the nonterminal hyperedges $E_p^N = \{e_1, ..., e_n\}$ of each production rule $p = (X, H)$ are in some (arbitrary, but fixed) linear order, say $e_1, ..., e_n$. For HRG G, $G[X]$ denotes the HRG (Σ_N, P_G, X).

Definition 6 (Derivation Tree). *Let $G = (\Sigma_N, P, S) \in \mathtt{HRG}$. The set of all* derivation trees *of G is the least set $\mathtt{D}(G)$ of trees over the alphabet P with ranking function $rk_P : P \to \mathbb{N}$ such that $t(\varepsilon) = p$ for some $p \in P$ with $\mathrm{lhs}(p) = S$. Moreover, if $E_p^N = \{e_1, \ldots, e_m\}$, then $rk_P(p) = m$ and $t|_i \in \mathtt{D}(G[\mathrm{lab}_p(e_i)])$ for each $i \in [m]$. The* yield *of a derivation tree is given by the HG*

$$\mathit{yield}(t) = t(\varepsilon)[e_1/\mathit{yield}(t|_1), \ldots, e_m/\mathit{yield}(t|_m)].$$

We implicitly assume that the nodes and hyperedges of $t(x)$ and $t(y)$ are disjoint if $x \neq y$. The yield of a derivation tree is also called the *derived* HG according to t.

Example 3. Figure 1(c) illustrates a derivation tree t of the HRG *TLL* in which production rule p_1 has been applied once, and production rule p_2 twice. The labels next to the circles provide the position in $\mathrm{dom}(t)$ while the labels inside indicate the applied production rule. The graph on the right (d) illustrates the shape of $\mathit{yield}(t)$. For simplicity, node indices as well as edge labels are omitted.

The language generated by an HRG consists of all HGs without nonterminal edges that can be derived from the initial nonterminal symbol.

Definition 7 (HR Language). *The language generated by $G \in \mathtt{HRG}$ is the set $L(G) = \{\mathit{yield}(t) \mid t \in \mathtt{D}(G)\}$.*

Example 4. The HRG *TLL*, provided in Figure 1, generates the set of all fully-branched binary trees in which the leaves are connected from left to right and each node has an additional edge to its parent.

Two results for derivation trees are needed in the following. The first result is directly lifted from analogous results for context-free word grammars (cf. [22] below Theorem 3.10).

Lemma 3 *For each $G \in \mathtt{HRG}$, $\mathtt{D}(G)$ is a regular tree language. In particular, the emptiness problem for HRGs is decidable in linear time.*

Furthermore, we generalize the notion of merged nodes to multiple successive applications of hyperedge replacement.

Definition 8 (Merged Nodes). *Let $G \in \mathtt{HRG}$, $t \in \mathtt{D}(G)$, $x, y \in \mathrm{dom}(t)$ such that $x \prec y$, i.e. $y = x.i.z$ for some $i \in \mathbb{N}, z \in \mathbb{N}^\star$, and let $u \in V_{t(x)}$, $v \in V_{t(y)}$. We say that u and v are merged in t, written $u \sim_t v$, if*

- *$z = \varepsilon$ and $u \;{}_{t(x),e_i}{\approx}_{t(x.i)}\; v$, or*
- *$z \neq \varepsilon$ and there exists $w \in V_{t(x.i)}$ such that $u \;{}_{t(x),e_i}{\approx}_{t(x.i)}\; w$ and $w \sim_t v$.*

Example 5. Consider the derivation tree t shown in Figure 1(c) again. In its yield, the node u_7 in $t(\varepsilon)$ is merged with u_4 in $t(1)$ and with u_3 in $t(2)$. In $\mathit{yield}(t)$, this node represents the leftmost leaf of the right subtree.

The relation $\sim_t$ merges exactly the nodes that are identified with each other by $\mathit{yield}(t)$.

Lemma 4 (Merge Lemma) *Given $G \in \mathtt{HRG}$ and $t \in \mathtt{D}(G)$, let $\simeq_t$ denote the least equivalence relation containing $\sim_t$. Then*

$$\mathit{yield}(t) \cong \left[\biguplus_{x \in \mathrm{dom}(t)} \mathrm{rhs}(t(x)) \right]_{/\simeq_t} .$$

4 Tree-Like Grammars

This section introduces tree-like grammars (TLG), a fragment of HRGs which can be translated into $\mathtt{MSO}_2$.

Some further notation is needed. Let $H \in \mathtt{HG}_{\Sigma_N}$ with $E_H^N = \{e_1, \ldots, e_m\}$. We call $\mathrm{ext}_H(1)$ the *anchor* node of H and denote it by $⚓_H$. Moreover, the sequence of *context* nodes of H is defined as $\mathrm{ctxt}_H := \mathrm{att}_H(e_1)(1) \ldots \mathrm{att}_H(e_m)(1)$ and the *free* nodes of H are all nodes attached to nonterminal hyperedges only, i.e. $\mathrm{free}(H) := \{u \in V_H \mid \forall e \in E_H^\Sigma : u \notin \lfloor \mathrm{att}_H(e) \rfloor\}$.

We will see that TLGs are constructed such that every anchor node u represents an application of a production rule and thus a position in a derivation tree t. The context nodes represent its children as they are merged with anchor nodes after their corresponding nonterminal hyperedges have been replaced. Consequently, by the characteristic edges of an anchor node u we refer to the characteristic edges $E_{t(x)}^\Sigma$ of a position $x \in \mathrm{dom}(t)$ represented by u. We consider a series of simple graph languages to narrow down the class of TLGs step by step. The first example stems from the fact that every context-free word language can be generated by an HRG [14] (if words are canonically encoded by edge-labelled graphs).

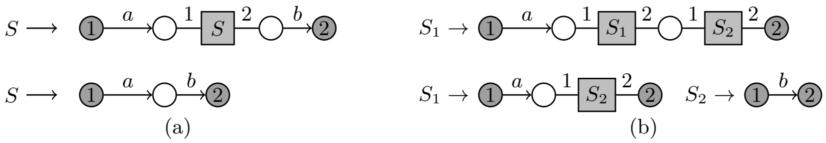


Fig. 2. Two HRGs generating the language $\{a^n.b^n \mid n \geq 1\}$ of string-like graphs.

Example 6. The HRG G shown in Figure 2(a) generates string-like graphs of the form $a^n.b^n$ for each $n \geq 1$. It is well known that the language $L(G)$ is not $\mathtt{MSO}_2$-definable. We observe that for arbitrary hypergraphs $H \in L(G)$ it is not possible to determine a node that is uniquely associated with all terminal edges in the recursive, upper production rule of Figure 2(a) (which is in accordance with the idea behind TLGs formulated at the beginning of this section). This is caused by the intermediate nonterminal hyperedge, which can be replaced by an arbitrarily large HG. Thus, to ensure that TLGs generate $\mathtt{MSO}_2$-definable hypergraphs only, we require that every non-free node (and thus every terminal edge) is reachable from the anchor node using terminal edges only.

However, this requirement is insufficient. For instance, Figure 2(b) depicts an HRG G' with $L(G') = L(G)$ which satisfies the condition from above. G' is obtained by transforming G into the well-known Greibach normal form (for word grammars). In a derivation tree t, a position $x \in \mathrm{dom}(t)$ corresponding to an application of the upper production rule has two children which represent the nonterminal hyperedges labelled with S_1 and S_2, respectively. Since all nodes except for the two leftmost ones are free in this production rule, the parent-child relationship between anchor nodes and context nodes (or any other triple of nodes) cannot be reconstructed in $\mathtt{MSO}_2$. Thus, we additionally require context nodes to be non-free.

In the following we consider *basic* tree-like HGs, which form the building blocks of which a tree-like HG is composed.

Definition 9 (Basic Tree-Like Hypergraphs). *$H \in \mathtt{HG}_{\Sigma_N}$ is a basic tree-like HG if $\text{⚓}_H \in \lfloor \mathrm{att}_H(e) \rfloor$ for each $e \in E_H^{\Sigma}$ and $\lfloor ctxt_H \rfloor \cap free(H) = \emptyset$.*

As a first condition on TLGs, we require right-hand sides of production rules to be (basic) tree-like. In case of string-like graphs, this condition is sufficient to capture exactly the regular word languages (if the direction of edges is ignored), because every such grammar corresponds to a right-linear grammar. If arbitrary graphs are considered, however, there are more subtle cases.

Example 7. Figure 3 (left) depicts an HRG G with three production rules p, q, r. $L(G)$ is the set of "doubly-linked even stars", i.e. a single node u connected by an incoming and an outgoing edge to each of $2n$ nodes for some $n \geq 0$. An HG $H \in L(G)$ is illustrated in Figure 3 (right). Again, $\mathrm{L}(G)$ is not $\mathtt{MSO}_2$-definable. In particular, no derivation tree can be reconstructed from H by identifying nodes (or edges) in H with positions in a derivation tree, because $|V_H| = 5$ and $|E_H| = 8$, but $|\mathrm{dom}(t)| = 9$. The problem emerges from the fact that all anchor nodes are merged with the central node u. Hence, we additionally require that anchor nodes are never merged with each other.

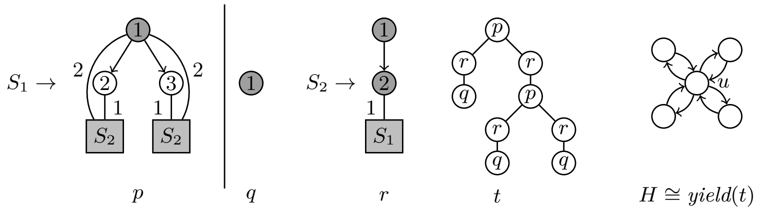


Fig. 3. An HRG G where production rules p, q, r map to tree-like HGs (left) and a generated graph $H \in L(G)$ (right)

Formally, for any $X \in N$, $H \in L(G[X])$ contains merged anchor nodes if for some $t \in \mathtt{D}(G[X])$ with $H \cong yield(t)$, there exist $x, y \in \mathrm{dom}(t)$, $x \neq y$ such that $\text{⚓}_{t(x)} \simeq_t \text{⚓}_{t(y)}$. The set of all HGs in $\bigcup_{X \in N} L(G[X])$ containing merged anchor nodes is denoted by $\mathcal{M}(G)$.

Definition 10 (Tree-Like Grammar). *$G = (\Sigma_N, P, S) \in \mathtt{HRG}$ is a TLG if $\mathcal{M}(G) = \emptyset$ and for each $p \in P$, $\mathrm{rhs}(p)$ is a basic tree-like HG. The set of all TLGs is denoted by $\mathtt{TLG}$.*

The condition $\mathcal{M}(G) = \emptyset$ is, admittedly, not syntactic. However, it is possible to automatically derive an HRG generating exactly the graphs satisfying it.

Theorem 2. *For each HRG G, one can construct a TLG G' such that $L(G') = L(G) \setminus \mathcal{M}(G)$.*

Remark 1. We call an HG H *tree-like* if it can be composed from basic tree-like HGs, i.e. there exists a TLG G with $L(G) = \{H\}$ (where nonterminals of H are considered to be terminal). Although only basic tree-like HGs are considered in all proofs, our results also hold for tree-like HGs. In particular, if all non-free nodes of an HG H are reachable from the anchor node without visiting an external node, a context node or a nonterminal hyperedge, H is tree-like. Intuitively, the anchor nodes of corresponding TLG production rules are determined by a spanning tree with the anchor of H as root. Analogously, the initial nonterminal S may be mapped to an arbitrary HG provided that it never occurs on the right-hand side of a production rule.

Example 8. According to the previous remark, the recurring example HRG *TLL* illustrated in Figure 1 is a TLG.

As already stated in the introduction, our main result is the following.

Theorem 1. *For each TLG G, there exists an $\mathtt{MSO}_2$ sentence φ_G such that for each $H \in \mathtt{HG}$, $H \in L(G)$ if and only if $\underline{H} \models \varphi_G$.*

An important observation to show this theorem is that every graph H generated by a TLG G has two properties:

1. A derivation tree t of H is $\mathtt{MSO}_2$-definable in H, i.e. TLGs generate recognisable graph languages in the sense of Courcelle [8].
2. Every edge $e \in E_H$ can be uniquely associated in $\mathtt{MSO}_2$ with some $x \in \mathrm{dom}(t)$ corresponding to the production rule $t(x)$ which added e to H.

Hence, given $\mathtt{MSO}_2$ formulae encoding t in H and defining $E^{\Sigma}_{t(x)}$ for each $x \in \mathrm{dom}(t)$, one can easily obtain a formula φ ensuring that all edges in every model of φ are edges introduced by the proper application of a production rule. In particular, $K = \biguplus_{x \in \mathrm{dom}(t)} \mathrm{rhs}(t(x))$ is a model of φ for each $t \in \mathtt{D}(G)$. By Lemma 4, it is sufficient to extend φ to an $\mathtt{MSO}_2$ sentence φ' such that only graphs H with $H \cong [K]_{/\simeq_t} \cong \mathit{yield}(t)$, i.e. graphs that resulted from hyperedge replacement steps where exactly the $[K]_{/\simeq_t}$-equivalent nodes were merged, are models of φ'. For any given pair of nodes, this property can be verfied by a finite (string) automaton running on a path in the derivation tree t.

We collect two direct consequences of Theorem 1 and Lemma 1.

Theorem 3. *The class of languages generated by TLGs is closed under union, intersection and difference.*

Theorem 4. *Given $G \in \mathtt{TLG}$ and $G' \in \mathtt{HRG}$, it is decidable whether $L(G') \subseteq L(G)$. In particular, the inclusion problem for TLGs is decidable.*

5 Tree-Like Separation Logic

As can be seen in Lemma 2, there exists a strong correspondence between $\mathtt{SL}$ and HRGs. This correspondence leads to portability of the obtained TLG results to analogous $\mathtt{SL}$ results. As $\mathtt{SL}$ is tailored to reason about heaps, we restrict ourselves to *data structure grammars* (DSG), i.e. HRGs generating heaps only. We denote the class of all DSGs by $\mathtt{DSG}$.

The largest $\mathtt{SL}$ fragment considered in this paper is $\mathtt{SL}_{\mathtt{tl}}$ as defined in the introduction (see Definition 1).

Theorem 5. *For every $\mathtt{SL}_{\mathtt{tl}}$ formula φ there exists a language-equivalent tree-like DSG G and vice versa.*

Example 9. Consider the $\mathtt{SL}_{\mathtt{tl}}$ formula $\varphi := \sigma(x_1, x_2, x_3, x_4)$ defined over an environment Γ consisting of predicate definitions for two predicate symbols σ and γ.

$$\begin{aligned}\sigma(x_1,x_2,x_3,x_4) &:= [\exists x_5,x_6,x_7 : x_1.(p,l,r) \mapsto (x_2,x_5,x_6) * \sigma(x_5,x_1,x_3,x_7)\\ &\quad * \sigma(x_6,x_1,x_7,x_4)] \vee [\exists x_5 : x_1.(p,l,r) \mapsto (x_2,x_3,x_5)\\ &\quad * x_3.p \mapsto x_1 * x_5.p \mapsto x_1 * \gamma(x_5,x_3,x_4)]\\ \gamma(x_1,x_2,x_3) &:= x_2.n \mapsto x_1 * x_1.n \mapsto x_3\end{aligned}$$

Applying Lemma 2 to φ and Γ yields a tree-like DSG generating the same language as the HRG *TLL* shown in Figure 1, i.e. the set of all fully-branched binary trees with linked leaves and parent pointers. In particular, the first disjunct of $\sigma(x_1, x_2, x_3, x_4)$ directly corresponds to the production rule in Figure 1(a), where variable names match with node indices. The other two disjuncts, split across two predicates, translate into basic tree-like HGs and correspond to the second production rule.

We can exploit the additional requirements for DSGs to obtain a simple, yet expressive, *purely syntactical* fragment of TLGs.

Definition 11 (Δ-DSGs). *Let $\Delta \subseteq \Sigma$ be a nonempty set of terminal symbols. Then $G = (\Sigma_N, P, S) \in \mathtt{DSG}$ is a Δ-DSG if for each $p \in P$, $\mathrm{rhs}(p)$ is a tree-like hypergraph and $⚓_p$ has an outgoing edge labelled δ for each $\delta \in \Delta$.*

Example 10. Our example HRG *TLL* shown in Figure 1 is a $\{p, l, r\}$-DSG.

Lemma 5 *Every Δ-DSG with $\emptyset \neq \Delta \subseteq \Sigma$ is a TLG.*

In terms of expressiveness, we may compare Δ-DSGs to $\mathtt{SL}_{\mathtt{btw}}$ [16], which is, to the best of our knowledge, the largest known fragment of $\mathtt{SL}$ with a decidable entailment problem. In particular, consider the $\{h\}$-DSG G depicted in Figure 4 generating reversed binary trees with an additional pointer to the head of another data structure. The language generated by G is not $\mathtt{SL}_{\mathtt{btw}}$-definable, because the number of allocated locations from which the whole heap must be reachable is fixed a priori for every $\mathtt{SL}_{\mathtt{btw}}$ formula and a corresponding environment.

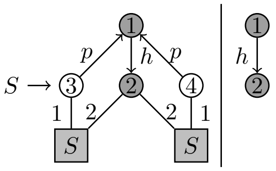


Fig. 4. Tree-like DSG

Theorem 6. *Δ-DSGs are strictly more expressive than $\mathtt{SL}_{\mathtt{btw}}$, i.e. for every $\mathtt{SL}_{\mathtt{btw}}$ formula there exists a language-equivalent Δ-DSG, but not vice versa.*

6 Conclusion

SL and DSGs are established formalisms to describe the abstract shape of dynamic data structures. A substantial fragment of SL is known to coincide with the class DSG. With this relationship, decidability of the satisfiability problem for SL, for instance, follows directly from decidability of its graph theoretic counterpart, the emptiness problem for DSGs. However, the entailment problem or, equivalently, the inclusion problem is undecidable.

$$\begin{tikzcd}
\mathtt{SL}_{\mathtt{RD}} \arrow[dd] & \mathtt{HRG} \arrow[d] \arrow[dr] & \mathtt{MSO}_2 \arrow[d] & \\
 & \mathtt{DSG} \arrow[dr] & \mathtt{TLG} \arrow[r] \arrow[d] & \mathtt{RGG} \\
\mathtt{SL} \arrow[ur, equal] \arrow[r] \arrow[d] & \mathtt{SL}_{\mathtt{tl}} \arrow[r, equal] \arrow[dl] & \mathtt{TL}-\mathtt{DSG} \arrow[d] & \\
\mathtt{SL}_{\mathtt{btw}} & & \Gamma-\mathtt{DSG} \arrow[ll] &
\end{tikzcd}$$

Fig. 5. Relationships between fragments of HRG and SL

We introduced the class TLG of tree-like grammars which generate $\mathtt{MSO}_2$ definable languages only. From this, some remarkable properties, like decidability of the inclusion problem and closure under intersection, directly follow by previous work on context-free and recognisable graph languages. Moreover, the close correspondence between HRGs and SL yields several fragments of SL, in particular $\mathtt{SL}_{\mathtt{tl}}$, where an extended entailment problem is decidable. The resulting fragments are more expressive than $\mathtt{SL}_{\mathtt{btw}}$, the largest fragment of SL with a decidable entailment problem known so far.

Figure 5 depicts an overview of the SL and HRG fragments considered in this paper, where an edge from formalism F_1 to formalism F_2 denotes that the class of languages realizable by F_2 is included in the class of languages realizable by F_1. All of these inclusion relations are strict. For completeness, we also added the class $\mathtt{SL}_{\mathtt{RD}}$ of completely unrestricted Separation Logic with inductive predicate definitions (cf. [16, 1]) and the class RGG of regular graph grammars [9].

With regard to future research, investigating decision procedures and their tractability for the entailment problem for (fragments of) $\mathtt{SL}_{\mathtt{tl}}$ is of great interest. Although the entailment and inclusion problem is effectively decidable for the fragments presented in this paper, our reliance on Courcelle's theorem does not lead to efficient algorithms (see [19] for a recent survey of alternative approaches). Still, a better understanding of the boundary between decidability and undecidability of the entailment problem for SL and the inclusion problem for HRGs might help to obtain more efficient algorithms for specialised fragments. We hope that a combined approach - studying SL as well as context-free graph languages - will lead to further improvements in this area.

References

1. Antonopoulos, T., Gorogiannis, N., Haase, C., Kanovich, M.I., Ouaknine, J.: Foundations for decision problems in separation logic with general inductive predicates. In: FOSSACS. Volume 8412 of LNCS. (2014) 411–425
2. Berdine, J., Calcagno, C., O'Hearn, P.W.: A decidable fragment of separation logic. In: FSTTCS. Volume 3328 of LNCS. (2005) 97–109

3. Berdine, J., Cook, B., Ishtiaq, S.: SLAyer: Memory safety for systems-level code. In: CAV. Volume 6806 of LNCS. (2011) 178–183
4. Brotherston, J., Distefano, D., Petersen, R.L.: Automated cyclic entailment proofs in separation logic. In: CADE-23. Volume 6803 of LNCS. (2011) 131–146
5. Brotherston, J., Kanovich, M.: Undecidability of propositional separation logic and its neighbours. In: LICS. (2010) 130–139
6. Büchi, J.R.: Weak second-order arithmetic and finite automata. Mathematical Logic Quarterly **6**(1-6) (1960) 66–92
7. Chin, W.N., David, C., Nguyen, H.H., Qin, S.: Automated verification of shape, size and bag properties via user-defined predicates in separation logic. Science of Computer Programming **77**(9) (2012) 1006–1036
8. Courcelle, B.: The monadic second-order logic of graphs I: Recognizable sets of finite graphs. Information and Computation **85**(1) (1990) 12–75
9. Courcelle, B.: The monadic second-order logic of graphs V: On closing the gap between definability and recognizability. Theoretical Computer Science **80**(2) (1991) 153–202
10. Courcelle, B., Engelfriet, J.: Graph Structure and Monadic Second-Order Logic: A Language-Theoretic Approach. Number 138. Cambridge University Press (2012)
11. Dodds, M.: From separation logic to hyperedge replacement and back. In: ICGT. Volume 5214 of LNCS. (2008) 484–486
12. Drewes, F., Kreowski, H.J., Habel, A.: Hyperedge replacement graph grammars. In: Handbook of Graph Grammars and Computing by Graph Transformation. (1997) 95–162
13. Dudka, K., Peringer, P., Vojnar, T.: Predator: A practical tool for checking manipulation of dynamic data structures using separation logic. In: CAV. Volume 6806 of LNCS. (2011) 372–378
14. Habel, A.: Hyperedge Replacement: Grammars and Languages. Volume 643 of LNCS. (1992)
15. Heinen, J., Noll, T., Rieger, S.: Juggrnaut: Graph grammar abstraction for unbounded heap structures. ENTCS **266** (2010) 93–107
16. Iosif, R., Rogalewicz, A., Simacek, J.: The tree width of separation logic with recursive definitions. In: CADE-24. Volume 7898 of LNCS. (2013) 21–38
17. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: Verifast: A powerful, sound, predictable, fast verifier for C and Java. In: NFM. Volume 6617 of LNCS. (2011) 41–55
18. Jansen, C., Göbe, F., Noll, T.: Generating inductive predicates for symbolic execution of pointer-manipulating programs. In: ICGT. Volume 8571 of LNCS. (2014) 65–80
19. Langer, A., Reidl, F., Rossmanith, P., Sikdar, S.: Practical algorithms for MSO model-checking on tree-decomposable graphs. Computer Science Review **13-14** (2014) 39–74
20. Lee, O., Yang, H., Yi, K.: Automatic verification of pointer programs using grammar-based shape analysis. In: ESOP. Volume 3444 of LNCS. (2005) 124–140
21. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS. (2002) 55–74
22. Salomaa, A., Rozenberg, G.: Handbook of Formal Languages, Vol. 3: Beyond Words. Springer (1997)

Towards data-flow oriented work flow systems

Alexander Mattes[1], Annette Bieniusa[1], and Arie Middelkoop[2]

[1] University of Kaiserslautern
[2] vwd Group

Abstract. Work flow frameworks have become an integral part of modern information systems. They provide a clearly defined interface and structure for interaction with a system. The specification of work flow systems, however, is usually ad-hoc. Often, programmers simply define a number of tasks (forms and actions) that are sequentially connected. Users are then restricted to input data in this prescribed order.
In this paper, we propose a data-flow oriented work flow system where the data flow is described by a purely functional program. This approach offers the user and the system flexibility in the order of tasks while guaranteeing a consistent and correct result.

1 Motivation

Work flow frameworks are a common component in today's content management systems. They enable the modeling, structuring, support, and execution of business processes. In this paper, we focus on a particular type of interactive work flow, called case management [1], that dictates the tasks a user needs to perform with the system (e.g. fill in a form, print a document, acknowledge some information) to complete some business process. This could range from surveys or tax forms to complexly structured financial consultation sessions.

This particular type of work flow is typically described with control flow graphs (e.g. Apache ODE). The programming of such work flows and their tasks fits neatly the imperative paradigm. However, such an approach forces the user to perform the tasks in a rigid predefined order, even when this is not strictly necessary to stay in compliance with the company's policies and local laws.

For example, in financial or insurance consultation, the sessions are desirably dynamic, with the consultant jumping back and forth through tasks, possibly revising prior inputs, depending on the interaction with the client. Such behavior is difficult to describe cleanly with control flow graphs, especially when it contains dynamic elements like loops.

Therefore, we propose a functional approach, in which the work flows are described with data flow graphs instead. The nodes of this graphs represent tasks (effectively idempotent impure functions with localized side effects) and the edges represent pure functions that map outputs of tasks to inputs of (other) tasks. Arrows [4] provide the necessary scaffolding for the (dynamic) construction of such graphs.

Two extensions of arrows are particularly important in this context: *monadic arrows* allow the structure of the work flow to be dependent on inputs (e.g. to generate a set of tasks for each client), and *feedback loops* model destructive changes (e.g. finalizing or closing some tasks).

We present these ideas in a simplified setting as an embedded domain specific language using Haskell. We start with an example that demonstrates the notation, then discuss its properties and its implementation in more detail.

2 Example

As an example, we consider a simple work flow consisting of three individual tasks, given by the data flow chart in Figure 1. The tasks `askName` and `askAge` ask for the name and age of the user, respectively. The task `askAddress` asks for the address of the user if he is of full age, otherwise it asks for the address of a parent or legal guardian. Therefore it depends on the return value of `askAge`.

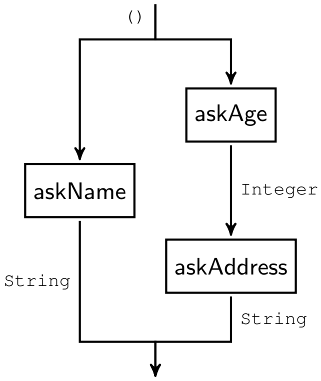


Fig. 1. Data flow of the example workflow.

While the data flow is fixed, the order in which the tasks have to be processed is only partially determined by the data dependencies. The only restriction is that the `askAge` task is executed before the `askAddress` task. Instead of programming the control flow by hand, we want to define the individual tasks independently of each other using pure and impure Haskell functions:

```
askName :: Task () String
askName = ...

askAge :: Task () Integer
askAge = ...

askAddress :: Task Integer String
askAddress = ...
```

In a second step, we combine them using `Arrow` constructors to model the data flow given by the chart:

```
workflow :: Task () (String,String)
workflow = askName &&& (askAge >>> askAddress)
```

Afterwards, it should be possible to execute the complete work flow using an automatically computed dynamic ordering of the tasks.

```
> runTask workflow ()
...
("Alice","Berlin")
```

where

```
runTask :: Task a b -> a -> IO b
runTask = ...
```

3 Implementation

In this section, we discuss the details of the task data type, the arrow combinators and execution of (composed) tasks in our Haskell implementation in detail.

3.1 Type Definition

We define a new parametric data type `Task a b` to model tasks with input type `a` and output type `b`. Since this type has to represent pure and impure tasks and support the various ways in which arrows can be combined, we need multiple constructors. In addition, we use a *generalized algebraic data type*(GADT) to achieve the necessary flexibility when composing tasks.

```
data Task a b where
    Pure     :: (a -> b) -> Task a b
    Impure   :: (a -> IO b) -> Task a b
    Serial   :: Task a b -> Task b c -> Task a c
    Parallel :: Task a b -> Task c d -> Task (a,c) (b,d)
```

The intended usage for the constructors is as follows:

`Pure` and `Impure` create single tasks out of pure functions and `IO` actions, respectively. While the `Impure` constructor would be sufficient for all functions (by simply lifting pure functions into the IO monad), this distinction allows us to later optimize the execution of the task; since pure functions have no side effects, the execution order is not important. In fact, we can rely on lazy evaluation to only compute those tasks whose results are actually needed.

`Serial` allows to compose two tasks in series, using the output of the first task as the input for the second. This dependency has to be considered later in the evaluation of the task. The `Serial` constructor basically models the composition `(>>>)` of arrows.

`Parallel` represents two tasks which are independent of each other and can thus be evaluated in an arbitrary order. It also represents the composition of two arrows with the `(***)` function.

3.2 Instance Declarations

The next step is to write an instance for the `Arrow` type class as well as for various subclasses. First, we have to turn `Task` into an instance of `Category`:

```
instance Cat.Category Task where
    id  = Pure id
    t1 . t2 = Serial t2 t1
```

The identity in `Catecory` is simply the pure task consisting of the identity function. The composition of two tasks is done with the `Serial` constructor.

Next, we can turn `Task` into an instance of the `Arrow` type class.

```
instance Arrow Task where
    arr       = Pure
    t1 *** t2 = Parallel t1 t2
    t1 &&& t2 = Pure (\a -> (a,a)) >>> t1 *** t2
    first t   = t *** Cat.id
    second t  = Cat.id *** t
```

The functions `arr` and `(***)` use the corresponding constructors `Pure` and `Parallel`. The `(&&&)` operator is implemented by using a pure function to feed the same input into both parallel tasks. `first` and `second` then simply use `(***)` and the identity task.

3.3 Additional Instances

To use the full potential of arrows, we can also create instances for the various arrow subclasses. For example, `ArrowChoice` allows the case distinction of two arrows based on the return value of the previous arrows and can be easily implemented by adding an additional constructor:

```
    Or       :: Task a c -> Task b c -> Task (Either a b) c
```

The instantiation of `ArrowChoice` is based on using `Or` to represent the `(|||)` operator.

```
instance ArrowChoice Task where
  (|||)    = Or
  f +++ g = (f >>> arr Left) ||| (g >>> arr Right)
    left f  = f +++ Cat.id
    right f = Cat.id +++ f
```

Similarly, its possible to create instances for other subclasses like `ArrowApply` or `ArrowLoop`, if necessary, by adding further constructors to the data type.

3.4 Optimizing the Instances

The given instances can be optimized to simplify the resulting data structures. Using pattern matching, we can for example introduce special cases for pure tasks. The composition of two pure tasks can be done with the creation of a pure task containing the composition of the functions which results in a simpler data structure:

```
    (Pure f1) . (Pure f2) = Pure (f1 . f2)
```

In the same way, the other arrow operators can be optimized for pure tasks by simply applying the operators to the contained functions and wrapping the result in a new pure task:

```
(Pure f1) *** (Pure f2) = Pure (f1 *** f2)
(Pure f1) &&& (Pure f2) = Pure (f1 &&& f2)
```

Notice that we cannot do the same with impure tasks since that would inevitably fix the execution order.

3.5 Arrow Laws

While `Task` is now technically an instance of the `Arrow` type class, we have to verify if it actually behaves like an arrow. As for many type classes, there are laws which every `Arrow` instance should obey [5]:

1. `arr id` = `id`
2. `arr (f >>> g)` = `arr f >>> arr g`
3. `first (arr f)` = `arr (first f)`
4. `first (f >>> g)` = `first f >>> first g`
5. `first f >>> arr fst` = `arr fst >>> f`
6. `first f >>> arr (id *** g)` = `arr (id *** g) >>> first f`
7. `first (first f) >>> arr assoc` = `arr assoc >>> first f`
   where `assoc ((a,b),c)` = `(a,(b,c))`

Note that there are similar laws for the `ArrowChoice` and `ArrowApply` type classes.

We want to check now if our implementation actually satisfies these laws. The first three are verified quite easily:

1.

```
arr id = Pure id
       = Category.id
```

2.

```
arr (f >>> g) = Pure (f >>> g)
              = Pure (g . f)
              = (Pure g) . (Pure f)
              = (arr g) . (arr f)
              = arr f >>> arr g
```

3.

```
first (arr f) = first (Pure f)
              = (Pure f) *** Cat.id
              = (Pure f) *** (Pure id)
              = (Pure f) *** (Pure id)
              = Pure (f *** id)
              = Pure (first f)
              = arr (first f)
```

For the fourth law, everything works as long as both tasks are pure:

4. (a) `f` = `Pure p` and `g` = `Pure q`:

```
first (f >>> g) = first (g . f)
                = first ((Pure q) . (Pure p))
                = first (Pure (q . p))
                = (Pure (q . p)) *** Cat.id
                = (Pure (q . p)) *** (Pure id)
                = Pure ((q . p) *** id)
                = Pure (first (q . p))
                = Pure (first (p >>> q))
                = Pure (first p >>> first q))
                = Pure ((p *** id) >>> (q *** id))
                = Pure (p *** id) >>> Pure (q *** id)
                = (Pure p) *** (Pure id) >>> (Pure q) *** (Pure id)
                = f *** Cat.id >>> g *** Cat.id
                = first f >>> first g
```

But if one of the tasks is not pure, the equality does not hold anymore:

(b) `f` ≠ `Pure p`:

```
first (f >>> g) = first (g . f)
                = first (Serial f g)
                = (Serial f g) *** Cat.id
                = Parallel (Serial f g) Cat.id
                ≠ Serial (f *** Cat.id) (Parallel g Cat.id)
                = Serial (f *** Cat.id) (g *** Cat.id)
                = Serial (first f) (first g)
                = first g . first f
                = first f >>> first g
```

The laws 5, 6 and 7 behave similarly. This means that only the first three laws hold in general; as soon as one of the tasks is not pure, they fail. This isn't surprising: For pure tasks, the laws are directly derivable from the fulfilled laws for the `Arrow` instance of functions. For all other tasks, the internal structure directly represents the order in which the arrows where combined. Hence, most laws have to fail. But is this a problem? The laws are there to guarantee that the instance actually behaves like an arrow. In our case, we are mostly interested in the behavior when we actually execute the task. It is therefore sufficient to check if the tasks are equivalent under execution.

3.6 Task execution

Until now, we considered the static representation of tasks as a composition of subtasks. Regarding their dynamic behavior, we needs a way to execute tasks. The `runTask` function executes a task as an IO action, thus allowing input and output of data:

```
runTask :: Task a b -> a -> IO b
```

The simplest way to do this is by remodeling the behavior of the *Kleisli arrow*.

```
runTask :: Task a b -> a -> IO b
runTask (Pure f)         = return . f
runTask (Impure m)       = m
runTask (Serial t1 t2)   = \a -> runTask t1 a >>= runTask t2
runTask (Parallel t1 t2) = \(u,v) -> do
                            r1 <- runTask t1 u
                            r2 <- runTask t2 v
                            return (r1,r2)
runTask (Or t1 t2)       = either (runTask t1) (runTask t2)
```

This certainly works, and it is also easy to show that this function is compatible with the arrow laws. For example, considering the fourth law,

$$\texttt{first (f >>> g)} = \texttt{first f >>> first g}$$

both sides of the equality evaluate to the execution of task `f` followed by the execution of task `g`.

While this solution respects the laws, it is not very useful yet. The execution order is fixed and not dynamic; in case of two parallel tasks, we always execute the left one first. To actually gain an advantage over simply using *Kleisli arrows*, we have to add some modifications. The easiest one would be to ask the user to specify the order in which parallel tasks get executed:

```
runTask (Parallel t1 t2) = \(u,v) -> do
                            putStrLn "Run Task a or b first?"
                            l <- getLine
                            if l == "b" then do
                                r2 <- (runTask t2) v
                                r1 <- (runTask t1) u
                                return (r1,r2)
                            else do
                                r1 <- (runTask t1) u
                                r2 <- (runTask t2) v
                                return (r1,r2)
```

This is an improvement but still not flexible enough. After choosing for example the left task, we have to finish all subtasks before we can start to execute the right task.

We can circumvent this problem by writing a function that executes only a single subtask:

```
runSingleTask :: (Task a b) -> a -> IO (Task a b)
runSingleTask t a = ...
```

We can then simply iterate over the subtasks of a composed task until all impure tasks have been processed:

```
runEveryTask :: (Task a b) -> a -> IO b
runEveryTask (Pure f) a = return . f $ a
runEveryTask t a = do
                s <- runSingleTask t a
                runEveryTask s a
```

To make this even more useful, we can add names

```
Impure   :: String -> (a -> IO b) -> Task a b
```

to the individual tasks to allow for an easier selection of which task to execute next.

A more advanced system could also offer the possibility to redo already processed tasks.

3.7 Example revisited

After all this work, we now return to our example from Section 2 and present an implementation. We start with the specification of the three in monadic syntax, assigning each a name:

```
askName :: Task () String
askName = Impure "askName" (const $ putStr "Please enter your name: " >>
     getLine)

askAge :: Task () Integer
askAge = Impure "askAge" (const $ putStr "Please enter your age: " >> readLn)

askAddress :: Task Integer String
askAddress = Impure "askAddress" $ \age -> do
        if age >= 18 then do
            putStr "Please enter your address: "
            getLine
        else do
            putStr "Please enter the address of a parent or legal guardian: "
            getLine
```

Then, we combine them with arrow operators reflecting the data dependencies to obtain the complete work flow:

```
workflow :: Task () (String,String)
workflow = askName &&& (askAge >>> askAddress)
```

When executing the tasks, we can dynamically choose the order in which the questions will be asked. For example, a session can take the following form:

```
> runEveryTask workflow ()
Open tasks:
0. askName
1. askAge
Which task do you want to run?: 1
Please enter your age: 23
Open tasks:
0. askName
1. askAddress
Which task do you want to run?: 0
Please enter your name: Alice
Please enter your address: Berlin
("Alice","Berlin")
```

4 Related work

The ideas in this paper are closely related to those of the iTasks system [6], which uses monads instead of arrows to describe work flows. Monads provide a means to describe a sequential work flow using pure functions. Recently, the authors seem to experiment with arrows as well [2]. Essential differences to our work are that we are not considering concurrent processes (orchestration), and instead enable different execution orders and revision of prior tasks.

These ideas are further a typical example of functional reactive programming [3], in particular with respect to the interaction of user and system.

5 Conclusion

In this paper, we presented a simple, but flexible work flow management framework. It allows to compose work flows from individual tasks and provides the means to process the subtasks in a dynamically adaptable order. A previous version of this work is incorporated in a real world financial application using a propriety functional programming language. Due to practical considerations, such as side effects in legacy code, that version used a mixture of control and data flow graphs. In this work, we essentially prototyped the next version which is based entirely on the work flow's data dependencies, and verified that the ideas are both viable and practical.

In future work, we will integrate our work flow management system into some web framework. This will provide programmers with the means to specify applications such as surveys and questionnaires in a simple and flexible way.

References

1. van der Aalst, W.M.P., Westergaard, M., Reijers, H.A.: Beautiful workflows: A matter of taste? In: Achten, P., Koopman, P.W.M. (eds.) The Beauty of Functional Code - Essays Dedicated to Rinus Plasmeijer on the Occasion of His 61st Birthday. Lecture Notes in Computer Science, vol. 8106, pp. 211–233. Springer (2013), http://dx.doi.org/10.1007/978-3-642-40355-2

2. Achten, P., van Eekelen, M.C.J.D., de Mol, M., Plasmeijer, R.: Editorarrow: An arrow-based model for editor-based programming. J. Funct. Program. 23(2), 185–224 (2013)
3. Hudak, P.: Principles of functional reactive programming. ACM SIGSOFT Software Engineering Notes 25(1), 59 (2000)
4. Hughes, J.: Generalising monads to arrows. Science of Computer Programming 37, 67–111 (May 2000), http://www.cs.chalmers.se/ rjmh/Papers/arrows.ps
5. Paterson, R.: "Control.Arrow". base-4.8.0.0: Basic libraries. haskell.org. Website (2002), retrieved 30 June 2015.
6. Plasmeijer, R., Achten, P., Koopman, P.W.M.: itasks: executable specifications of interactive work flow systems for the web. In: Hinze, R., Ramsey, N. (eds.) Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007. pp. 141–152. ACM (2007), http://doi.acm.org/10.1145/1291151.1291174

Towards a Theory of Objects in Sequentially Constructive Synchronous Programming

Michael Mendler and Marc Pouzet

[1] University of Bamberg
[2] Ecole Normale Superieure Paris

Abstract. The synchronous model of computation reduces the programming of deterministic concurrent systems to the programming of stateful reaction modules that operate in lock-step. At each macro-step, also called synchronous instant, each concurrent program module reads inputs from the environment and executes a step function to change internal memory and produce an output which is consumed by the environment during the same instant. To guarantee overall determinacy, current synchronous programming (SP) languages are heavily restrictive: Modules may only communicate through signals, the modules' step functions must be schedulable so that there is essentially only one write access to a signal and each step function is called at most once within a single instant. Programs which cannot be scheduled to satisfy this are considered non-constructive and rejected. Thus, on the face of it, the synchronous paradigm, as embodied in traditional SP languages, seems to preclude object-style component models, which are common in mainstream imperative programming and natural for modular compilation of synchronous programs.

Previous attempts to add objects to SP have been fairly tentative or remained hidden in the intermediate languages of SP compilers. However, the situation may now be changing. Recent work on a scheduling-centred reconstruction of SP, called the sequentially constructive model of synchronous computation (SCMoC), has introduced a key advance. The SCMoC permits multiple sequential writes to a signal variable under an init-update-read scheduling discipline which relaxes the standard constructiveness analysis for an SP program. Considering that a signal is nothing but a rather special case of a shared object, we show how to enrich earlier tentative synchronous object models by pushing the SCMoC scheduling perspective further. We generalise from simple read/write access functions on signals to module tasks on shared state, and from predefined implicit scheduling disciplines to programmer-defined scheduling policies. In this way, we can encapsulate both memory and synchronous code freely into shared objects just as SCMoC signals can be shared modulo init-update-read protocols. This yields an expressive component model that fills an abstraction gap still prevalent in standard SP languages.

Distribution Class Analysis

Hans Moritsch

Vienna University of Technology
Institute of Computer Languages
Argentinierstraße 8
A-1040 Vienna, Austria
hans.moritsch@tuwien.ac.at

Abstract. The compilation of languages for parallel computations which provide constructs to control the dynamic allocation of data structures to processors requires the analysis of the relationship between array references and data distribution functions. Expressions representing classes of distribution functions constitute the basic data flow information. For the interprocedural analysis, they are represented symbolically. A backward phase over the call graph calculates precise procedure summaries, a subsequent forward phase, by propagating calling context, eliminates the symbols therein. The meet operation allows for subsumptions, subclasses of already occurring classes are ignored. If the control flow is affected by the distribution class related to a reference, the data flow information is masked with the decisive condition. The analysis is based on a reduced flow graph containing only the nodes relevant to the analysis outcome.

1 Introduction

Information about the distribution of arrays is essential for a compiler for a data parallel programming language to generate efficient code. Optimization strategies can improve the code considerably, if detailed knowledge about the use of data and work distributions in a program is available. Extensive analysis has to provide this information.

The *distribution* of an array is a function that defines how the elements of the array are partitioned into subsets—often rectangular *segments*—and mapped to processors. This has various implications, most importantly, with respect to the allocation of arrays in the processors' memories, and the processors' responsibilities for preforming calculations with arrays.

The array distribution is specified in connection with the array declaration, after the keyword DIST, for each array dimension as a reference to an intrinsic distribution function such as BLOCK or CYCLIC, or the keyword NONE for "no distribution".[1] For example, a DIST (BLOCK BLOCK) specifies a blockwise distribution for array a in both dimensions, whereas b DIST (BLOCK NONE) defines, that array b is distributed blockwise in the first dimension, and not distributed

[1] NONE for every dimension specifies a replicated array, yet this is seen as a special case of distribution.

in the second dimension i.e., b is is partitioned into blocks of rows. Vice versa, (NONE BLOCK) specifies blocks of columns (see Figure 1).

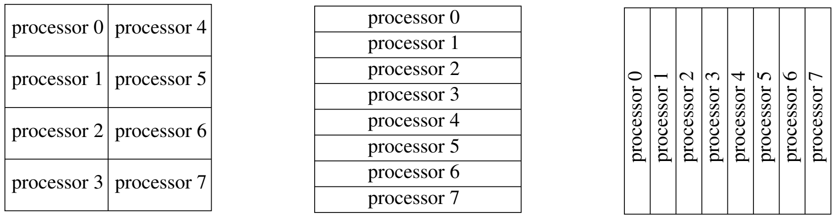


Fig. 1. Segments of a (BLOCK BLOCK), (BLOCK NONE), and (NONE BLOCK) distribution

The distribution of an array may be changed at runtime by means of the DISTRIBUTE statement. Through

DISTRIBUTE b (NONE BLOCK)

b will be *redistributed* to a (NONE BLOCK) distribution.

The current distribution of an array can influence the control flow in the program by means of a *distribution query* in the form of a test for "Identical Distribution Types".

if a IDT (BLOCK $*$) then . . . else . . . ,

restricts the execution of the then-part to states in which a is distributed blockwise in the first dimension, independent of its distribution in the second dimension. The complementary condition is relevant to the else-part. The wildcard "$*$" is used in the standard manner.

Whenever a distributed array is passed as a parameter in a procedure call, its distribution is passed as well. On procedure return, the actual parameter adopts the distribution of the formal parameter.[2]

The language constructs presented show, in a simplistic syntax, a small subset of the features for distributed arrays in data parallel languages [5, 4]. Primarily, they are supposed to serve as the basis for the presentation of our analysis.

This analysis delivers a range of possible distribution states for each statement of a program composed of several procedures. It includes an intra- and an interprocedural level, connected with each other through a symbolic representation of distribution classes (see Section 4, Section 3).

As a special feature, the analysis deals also with classes of distributions constituted by use of wildcards. This gives reasons for the definition of a *subsumption* relation between more general and more specific classes of distributions.

[2] A different behavior can be achieved through appropriate placements of DISTRIBUTE statements at procedure boundaries.

2 Intraprocedural Data Flow Analysis

2.1 Data Flow Information

A *distribution expression* is a list, equal in length to the number of array dimensions; each element corresponds to an array dimension and is

- an intrinsic distribution function name (such as BLOCK or CYCLIC),
- or NONE,
- or the wildcard "$*$", which stands for any of the above.

A distribution expression specifies a class of distributions, which is called a distribution type [5].

A *distribution state* is a vector of distribution expressions, in which each component is assigned to a declared (distributed) array. Alternatively, a component can be formed from an array identifier, namely of a formal parameter array, which accepts its distribution type from the corresponding actual parameter. The array identifier is in fact a symbol representing the transferred distribution type.

A *masked distribution state* is a distribution state with a distribution mask, a *distribution mask* is a set of pairs (array identifier, distribution expression). A pair may be labeled with a negation "$\neg$". The same array identifier can occur in more than one pairs, thus an arbitrary number, including zero, of distribution expressions can be assigned to an array. It is also possible, that no array occurs, i.e., the mask is the empty set. A distribution state without a (non-empty) mask is called *immediate*.

The data flow information set consists of *sets* of masked distribution states.

Example 1. The masked distribution state

$$\{a\,(\textsc{block}\ *), \neg\, c\,(\textsc{block}\ \textsc{none})\} \cdot [^{a:}(\textsc{block}\ \textsc{cyclic})\ ^{b:}\hat{a}\ ^{c:}(*\ \textsc{block})]$$

at a program point denotes the following.[3] If, on procedure entry, array a is BLOCK distributed in the first dimension, irrespective of its distribution in the second dimension, and array c is not (BLOCK NONE) distributed, then a is (CYCLIC BLOCK) distributed at that point, and array b has the same distribution type as a on procedure entry, and array c is BLOCK distributed in the second dimension, whereas its distribution in the first dimension is not specified and can be any of BLOCK, CYCLIC, NONE.[4]

A distribution expression is called *generic* in the presence of one or more wildcards.[5] A distribution expression which is not generic is called *terminal.* A distribution state is called generic, if it contains at least one generic distribution expression, otherwise it is called terminal.

[3] A pair (a, δ) in a mask is written as $a\,\delta$, and an array identifier component as $\hat{a}$.

[4] "a is δ distributed", "a has a δ distribution", and "a has the distribution type δ" have the same meaning.

[5] The genericity is an extension of the definition in [5].

2.2 Subsumption

Based on genericity, we define a relation on distribution expressions. A distribution expression δ is said to *subsume* a distribution expression δ', $\delta \succeq \delta'$, iff δ and δ' are equal in length, and δ contains $n \geq 0$ wildcards, and δ' contains at most n wildcards, and for each wildcard in δ' there is a wildcard in δ at the same position (i.e., for the same array dimension). By this definition a distribution expression subsumes itself. In contrast to this, a distribution expression δ is said to *properly subsume* a distribution expression δ', $d \succ \delta'$, iff $d \succeq \delta'$ and $\delta' \not\succeq d$. Note that $d \succ \delta'$ implies $d \neq \delta'$ and δ being generic.

Let $\tau(\delta)$ denote the set of terminal distribution expressions represented by δ. Iff δ subsumes δ', $\tau(\delta')$ is a subset of $\tau(\delta)$.

A distribution state s is said to subsume a distribution state s', $s \succeq s'$, iff s and s' are equal in length, and every component of s formed from a distribution expression subsumes the corresponding component of s', and every component formed from an array identifier is equal to the corresponding component of s'.

A distribution mask m specifies a predicate $S(m)$ on the distribution state of the formal parameter arrays on procedure entry. A distribution masks m is said to subsume a distribution mask m', $m \succeq m'$, iff m' represents the same or a stronger condition than m, i.e., iff $S(m') \subseteq S(m)$.

A masked distribution state $w = m \cdot s$ is said to subsume a masked distribution state $w' = m' \cdot s'$, $w \succeq w'$, iff $m = m'$ and $s \succeq s'$, or $m \succeq m'$ and $s = s'$, or both $m \succeq m'$ and $s \succeq s'$ hold, i.e., at least one of its parts subsumes the corresponding part of w'.

Sets of masked distribution states, which do not contain elements properly subsumed by other elements, are called *subsumption free.* They constitute the elements of the data flow information set L. Let W denote the set of masked distribution states. Then $L = \mathcal{P}(W)$, and for all $\ell \in L$, $w, w' \in W$

$$w \in \ell \wedge w \succ w' \rightarrow w' \notin \ell.$$

2.3 Meet Operation

The meet operation is a modified set union operation such that the result does not contain elements which are properly subsumed by other elements, the *subsumption free union* $\tilde{\cup}$. Let $\ell_1, \ell_2 \in L$. Then

$$\ell_1 \,\tilde{\cup}\, \ell_2 := \ell_1 \cup \ell_2 \, - \{w' \in W \mid \; \exists w \in \ell_1 \cup \ell_2 \wedge w \succ w' \}.$$

Based on $\tilde{\cup}$ a partial order on L can be defined,

$$\ell_1 \leq \ell_2 \Leftrightarrow \ell_1 \tilde{\cup} \ell_2 = \ell_1.$$

$\leq$ is the modified superset relation $\tilde{\supseteq}$,

$$\ell_1 \tilde{\supseteq} \ell_2 \Leftrightarrow \forall w' \in \ell_2 \, [\, w' \in \ell_1 \vee \exists w \in \ell_1 \; (w \succ w') \,].$$

2.4 Transfer Functions

The distribution state can change due to (i) redistributions, (ii) distribution queries, and (iii) procedure calls. For the pertaining types of flow graph nodes the transfer functions are given; for all other types the transfer function is the identity function.

In the following, let $s[a]$ denote the component of a distribution state s assigned to array a (the "a-component"), $w.m$ denote the mask part of a masked distribution state $w = m \cdot s$, $w.s$ denote the state part of w, and $\ell \in L$ denote the incoming data flow information at a node.

2.4.1 Redistribution The effect of the redistribution

$$\textsc{distribute}\ a\ \delta,$$

where a is an array and δ is a distribution expression, is defined by the transfer function $f^{\mathrm{D}} : L \to L$ of a distribute node (a node representing a DISTRIBUTE statement). It sets in all elements of ℓ the a-component to δ. I.e., $f^{\mathrm{D}}(\ell)$ executes, $\forall\, w \in \ell$,

$$w.s[a] \leftarrow d.$$

In case of a statement

$$\textsc{distribute}\ a = b,$$

where both a and b are arrays, it sets the a-component to the distribution expression, or the array identifier, which is currently assigned to array b. Thus $f^{\mathrm{D}}(\ell)$ executes, $\forall\, w \in \ell$,

$$w.s[a] \leftarrow w.s[b].$$

Example 2. The redistribution

$$\textsc{distribute}\ a\ (\textsc{block cyclic})$$

in the state

$$[^{a:}(\textsc{block block}),\ ^{b:}(\textsc{block none})]$$

yields

$$[^{a:}(\textsc{block cyclic}),\ ^{b:}(\textsc{block none})],$$

whereas

$$\textsc{distribute}\ a = b,$$

in the same state, yields

$$[^{a:}(\textsc{block none}),\ ^{b:}(\textsc{block none})].$$

2.4.2 Distribution Query

A node n in the flow graph representing a distribution query

$$\text{if } a \ \textsc{idt}\ \delta \text{ then } \ldots \text{ else } \ldots,$$

where a is an array and δ is a distribution expression, has two successor nodes, corresponding to the two possible outcomes. For processing distribution queries, prior to the analysis, two *assertion nodes*, n^{true} and n^{false}, are added to the flow graph between n and its successors.[6] The effect of the query is specified through the transfer function $f^{\mathrm{A}} : L \to L$ of an assertion node; the transfer function of n itself is the identity function.[7]

We have to distinguish, whether the a-component of a distribution state in ℓ, $w.s[a]$, is (i) a distribution expression η, or (ii) an array identifier $\hat{u}$.

In the first case, for n^{true}, the query is evaluated through determining the intersection $\delta \cap \eta$; for n^{false}, the intersection $\neg\delta \cap \eta$ is calculated, respectively. If the result is empty, the entire distribution state is removed from ℓ, otherwise the a-component is set to the intersection. So, $f^{\mathrm{A}}(\ell)$ for n^{true} executes, $\forall\, w \in \ell$,

$$w.s[a] \leftarrow \delta \cap w.s[a].$$

In the second case, as the query depends on $\hat{u}$, it cannot be evaluated. Its effect is expressed symbolically, through adding the pair $(\hat{u}, \delta)$ for n^{true} (the pair $(\hat{u}, \neg\delta)$ for n^{false}, respectively), to the mask. Thus $f^{\mathrm{A}}(\ell)$ for n^{true} executes, $\forall\, w \in \ell$,

$$w.m \leftarrow w.m \cup (w.s[a], \delta).$$

$f^{\mathrm{A}}(\ell)$ for n^{false} is specified analogously.

Example 3. The effect of the query

$$a \ \textsc{idt}\ (\textsc{block}\ *)$$

on the state (in the then-part)

$$[{}^{a:}(*\ \textsc{block}),\ {}^{b:}(\textsc{block none})]$$

is, with $(\textsc{block}\ *) \cap (*\ \textsc{block}) = (\textsc{block block})$,

$$[{}^{a:}(\textsc{block block}),\ {}^{b:}(\textsc{block none})].$$

In contrast, for the state

$$[{}^{a:}\hat{a}\ {}^{b:}(\textsc{block none})],$$

f^{A} yields

$$\{a\,(\textsc{block}\ *),\} \cdot [{}^{a:}\hat{a}\ {}^{b:}(\textsc{block none})].$$

[6] For queries without else-part, only what is said about n^{true} applies.

[7] However, f^{A} knows both a and δ.

2.4.3 Procedure Call A procedure call can, by some redistribution in the called procedure, change the distribution type of a parameter array. Further, the effect can, through a distribution query, depend on the current (i.e., before the call) distribution type of the same, or a different, actual parameter.

Transfer functions for procedure call and return nodes are involved in handling a procedure call. We assume only one return node in a procedure's flow graph.

Return Node The purpose of the transfer function $f^{\mathrm{R}} : L \to L$ for a return node of a procedure p is to summarize the effect of a call of p on the formal parameter arrays' distribution types. Local arrays in p do not affect the analysis of the calling procedure, hence f^{R} shrinks ℓ to information about formal parameter arrays. $f^{\mathrm{R}}(\ell)$ executes, $\forall\, w \in \ell$,

$$w.s \leftarrow \Pi_{F_p}(w.s),$$

where Π_{F_p} denotes the projection onto the set F_p of formal parameter arrays of p. Components of s assigned to local arrays, if they exist, are removed. Since only formal parameter arrays can occur in a mask, f^{R} has no effect on the latter.

After completion of the analysis of a procedure, R_p, the set of *return states*, identical with the result of f^{R}, describes the effect of a call.

Call Node The processing of a procedure call requires the previous analysis of the called procedure. The transfer function of the call node $f^{\mathrm{C}} : L \to L$ is based on R_p and has to interpret the parameter transfer. Masks in distribution states in ℓ, i.e., at the calling site, remain unaffected by f^{C}. In the following, let A denote the set of actual parameters of a call of p, and $\varphi_p : A \to F_p$ denote the mapping to corresponding formal parameters.[8]

Immediate Return States At first we consider distribution states in R_p without masks. $f^{\mathrm{C}}(\ell)$ executes, for $\forall\, w \in \ell, r \in R_p$, acting on a temporary copy w' of w,

$$\forall\, a \in A \colon t \leftarrow r.s[\varphi(a)],$$

where t is the component of r corresponding to the actual parameter a. If t is a distribution expression, then the component's new value is ready, so

$$w'.s[a] \leftarrow t.$$

Otherwise, t is an identifier of a formal parameter array. The corresponding actual parameter is $\varphi^{-1}(t)$, and the respective component at the call site will become the new value

$$w'.s[a] \leftarrow w.s[\varphi^{-1}(t)],$$

regardless of whether it is a distribution expression or an array identifier. In the latter case, it refers to a formal parameter of the calling procedure (currently analyzed) and thus represents a distribution type passed to it from *its* caller.

[8] We consider only array parameters.

Components of w' assigned to other arrays than actual parameters of the call remain unmodified. The ultimate result of f_p^{C} is formed from the subsumption free union of all temporary states w' built as described,

$$f_p^{\mathrm{C}}(\ell) = \widetilde{\bigcup}_{w \in \ell, r \in R_p} w'.$$

Example 4. The analysis of the procedure $p(x, y, z)$ reveals the return state

$$r = [^{x:}\,(\textsc{block block}),\ ^{y:}\,\hat{z},\ ^{z:}\,\hat{x}].$$

The call $p(a, b, c)$ in state

$$w = [^{a:}\,(\textsc{block cyclic}),\ ^{b:}\,(\textsc{block none}),\ ^{c:}\,\hat{b},\ ^{d:}\,(*\ \textsc{block})]$$

produces, by
$r.s[\varphi(a)] = r.s[x] = (\textsc{block block})$,
$r.s[\varphi(b)] = r.s[y] = \hat{z}$, $w.s[\varphi^{-1}(z)] = w.s[c] = \hat{b}$,
$r.s[\varphi(c)] = r.s[z] = \hat{x}$, $w.s[\varphi^{-1}(x)] = w.s[a] = (\textsc{block cyclic})$,
the state

$$w' = [^{a:}\,(\textsc{block block}),\ ^{b:}\,\hat{b},\ ^{c:}\,(\textsc{block cyclic}),\ ^{d:}\,(*\ \textsc{block})].$$

Example 5. The analysis of the procedure $q(x, y)$ reveals the return state

$$r = [^{x:}\,(\textsc{block cyclic}),\ ^{y:}\,\hat{y}].$$

The call $q(a, b)$ in the state

$$w = [^{a:}\,(\textsc{block block}),\ ^{b:}\,(\textsc{block none})]$$

produces, by
$r.s[\varphi(a)] = r.s[x] = (\textsc{block cyclic})$,
$r.s[\varphi(b)] = r.s[y] = \hat{y}$, $w.s[\varphi^{-1}(y)] = w.s[b] = (\textsc{block none})$,
the state

$$w' = [^{a:}\,(\textsc{block cyclic}),\ ^{b:}\,(\textsc{block none})].$$

In contrast, the call $q(a, b)$ in the masked state

$$w = \{a\,(\textsc{block}\ *)\} \cdot [^{a:}\,(\textsc{block block}),\ ^{b:}\,(\textsc{block none})]$$

produces, retaining the mask,

$$w' = \{a\,(\textsc{block}\ *)\} \cdot [^{a:}\,(\textsc{block cyclic}),\ ^{b:}\,(\textsc{block none})].$$

Masked Return States A mask in a return state can be evaluated to the extent in which it does not relate (taking into account the formal–actual parameter mapping) to distribution types transferred already to the calling procedure.

For every actual parameter, the distribution expression δ of every pair in the mask, in which the formal parameter corresponding to the actual parameter a occurs, will be compared with the a-component of the distribution state w

$$\forall\, a \in A \colon \forall\, (\varphi(a), \delta) \in m \colon t \leftarrow w.s[a].$$

If t is a distribution expression η, the pair is evaluated through computing the intersection $\delta \cap \eta$. Only if the result is non-empty for all pairs, w' will be produced as described, and added to the result of f_p^C.

If t is an array identifier $\hat{u}$, the pair $(\hat{u}, \delta)$ is added to the mask,

$$w'.m \leftarrow w'.m \cup (w.s[a], \delta).$$

This is equivalent to the handling of queries in f^A. However, here the condition arises from a query in the called, not in the currently analyzed, procedure.

Example 6. The analysis of the procedure $q'(x, y)$ reveals the masked return state

$$r = \{x\,(\textsc{block}\ *)\} \cdot [^{x\colon}(\textsc{block cyclic}),\ ^{y\colon}\hat{y}].$$

The call $q'(a, b)$ in the state

$$w = [^{a\colon}(\textsc{block block}),\ ^{b\colon}(\textsc{block none})]$$

evaluates $(\varphi(a), (\textsc{block}\ *)) \in r.m$, $w.s[a] = (\textsc{block block})$,
$(\textsc{block block}) \cap (\textsc{block}\ *) = (\textsc{block block}) \neq \varnothing$.

As the intersection is non-empty, f^C produces (see Example 5) the state

$$w' = [^{a\colon}(\textsc{block cyclic}),\ ^{b\colon}(\textsc{block none})].$$

In contrast, the call $q'(a, b)$ in the state

$$w = [^{a\colon}(\textsc{cyclic block}),\ ^{b\colon}(\textsc{block none})]$$

evaluates
$w.s[a] = (\textsc{cyclic block})$,
$(\textsc{cyclic block}) \cap (\textsc{block}\ *) = \varnothing$,
hence nothing will be produced.

Example 7. The call $q'(a, b)$ in the state

$$w = [^{a\colon}\hat{a}\ ^{b\colon}(\textsc{block none})]$$

evaluates $w.s[a] = \hat{a}$. The intersection $\hat{a} \cap (\textsc{block}\ *)$ is unfeasible, hence the result (see Example 6) will be equipped with an equivalent mask,

$$w' = \{a\,(\textsc{block}\ *)\} \cdot [^{a\colon}(\textsc{block cyclic})\ ^{b\colon}(\textsc{block none})].$$

If $q'(a, b)$ is called in the masked state

$$w = \{b\,(*\ \textsc{block})\} \cdot [^{a:}\,\hat{a}\ ^{b:}(\textsc{block none})],$$

the mask will be expanded,

$$w' = \{a\,(\textsc{block}\ *), b\,(*\ \textsc{block})\} \cdot [^{a:}(\textsc{block cyclic})\ ^{b:}(\textsc{block none})].$$

2.5 Flow Graph Reduction

Most often, the nodes of the flow graph actually having an effect on the analysis represent a very small fraction of all the nodes. It is obvious that excluding the irrelevant rest (nodes with identity transfer functions) from the analysis can significantly improve its performance. The flow graph can be reduced to the minimum extent necessary. By inserting an edge from an irrelevant successor k of a relevant node n_1 to every relevant node n_2 that can be reached from k along a path $(k, k_1, \ldots, k_l, n_2)$ of irrelevant nodes $k_i, i \geq 1$, the k_i can be eliminated. Subsequent to the analysis of a procedure, the data flow information at the reduced flow graph's node k is propagated to the nodes $(k_1, \ldots, k_l)$ in the original flow graph.

3 Symbolic Interprocedural Analysis

3.1 Backward Pass

In a (first) *backward* pass, the MASKED_DISTRIBUTION_STATES data flow analysis (see Section 2) is performed for each procedure, in reverse topological order along the call graph. Unknown distribution types handed over from calling procedures are represented by means of symbols (identifiers of formal parameters arrays, see Section 2.1). The result of the analysis of a procedure is expressed—without loss of precision—as a procedure summary (set of return states, see Section 2.4.3) using these symbols.

A called procedure is always analyzed in advance of the calling one(s), so it is possible to treat a call by interpretation of the procedure summary; there is no need to analyze the called procedure specifically for different actual parameter distribution types, or different call sites; every procedure needs to be analyzed only once.

3.2 Forward Pass

Information obtained in the analysis of the main procedure does not depend on calling context and therefore does not refer to symbols. Consequently, it does not contain masks, i.e., the result of the analysis of the main procedure is built from immediate states only. In a (second) *forward* pass, this information is gradually propagated into all called procedures where it allows for resolving the symbols, and thus eliminating the masks (see Algorithm 1).

For each procedure, in topological order along the call graph, and for all procedure calls therein, and for all—now immediate—states at such a call site (call node), the actual parameter distribution types are mapped to the formal parameters. In the called procedure, the—now known—distribution types of the formal parameters can be substituted for the symbols, hence all masks can be evaluated (cf. Section 2.4.3). A non-empty intersection for all pairs in a mask must appear at least once (over all states at all call sites), otherwise the state will eventually be removed.

The remaining states represent the result of the whole analysis.

Algorithm 1. Interprocedural Distribution Class Analysis

```
{backward pass:}
for each procedure $p$ in reverse topological order do
    solve MASKED_DISTRIBUTION_STATES on $p$'s flow graph
endfor

{forward pass:}
for each procedure $p$ in topological order do
    if $p \neq$ MAIN then
        remove masked states with untagged masks in $p$
        remove masks in $p$
    endif
    {all states in $p$ are immediate, propagate them into called procedures:}
    for each call of a procedure $q$ do
        for each state $s$ at call site do
            {evaluate masks in $q$:}
            for each mask $m$ in $q$ do
                if all pairs in $m$ yield non-empty then tag mask
            endfor
        endfor
    endfor
endfor
```

Figures 2 to 4 show the complete analysis of a program consisting of a main procedure which calls a procedure p, which in turn calls a procedure q.

4 Conclusion

In this paper, we presented an interprocedural data flow analysis which determines for every program statement the set of possible states of array distributions, considering classes of distributions. It deals with dynamic redistribution and the impact of distribution queries. The data flow information has a complex structure and supports subsumption relationships through wildcards.

The data flow analysis of procedure calls avoids approximations; the result is equivalent to the inline expansion of the calls. This is achieved by representing in the data flow information the formal parameters' properties as symbols, and by carrying along the conditions involving formal parameters' properties. In the second pass the symbols and conditions are resolved.

In our opinion, the basic principle behind is very general and can be employed for other kinds of data flow information and transfer functions than for the actual analysis as well.

References

1. B. Chapman, P. Mehrotra, H. Moritsch, and H. Zima. Dynamic Data Distributions in Vienna Fortran. *In Proceedings of the Supercomputing '93 Conference.* November 1993, Portland, Oregon.
2. B. Chapman, H. Moritsch, and H. Zima. Dynamically Distributed Arrays: Specification of the Compilation Method. Deliverable D1Z-2, CEI Project PACT - Programming Environments, Algorithms, Applications, Compilers, and Tools for Parallel Computation, April 1994.
3. B. Chapman, H. Moritsch, and H. Zima. The Implementation of Dynamic Data Distributions in the Vienna Fortran Compilation System: Language, Compile- and Run-Time Support. Deliverable D5.1f, ESPRIT III Project PPPE - Portable Parallel Programming Environments, July 1995.
4. High Performance Fortran Forum. *High Performance Fortran Language Specification, Version 2.0*, January 1997.
5. H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran – A Language Specification. ICASE Internal Report 21, ICASE, Hampton VA, 1992.
6. H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press Frontier Series, Addison-Wesley, 1990.

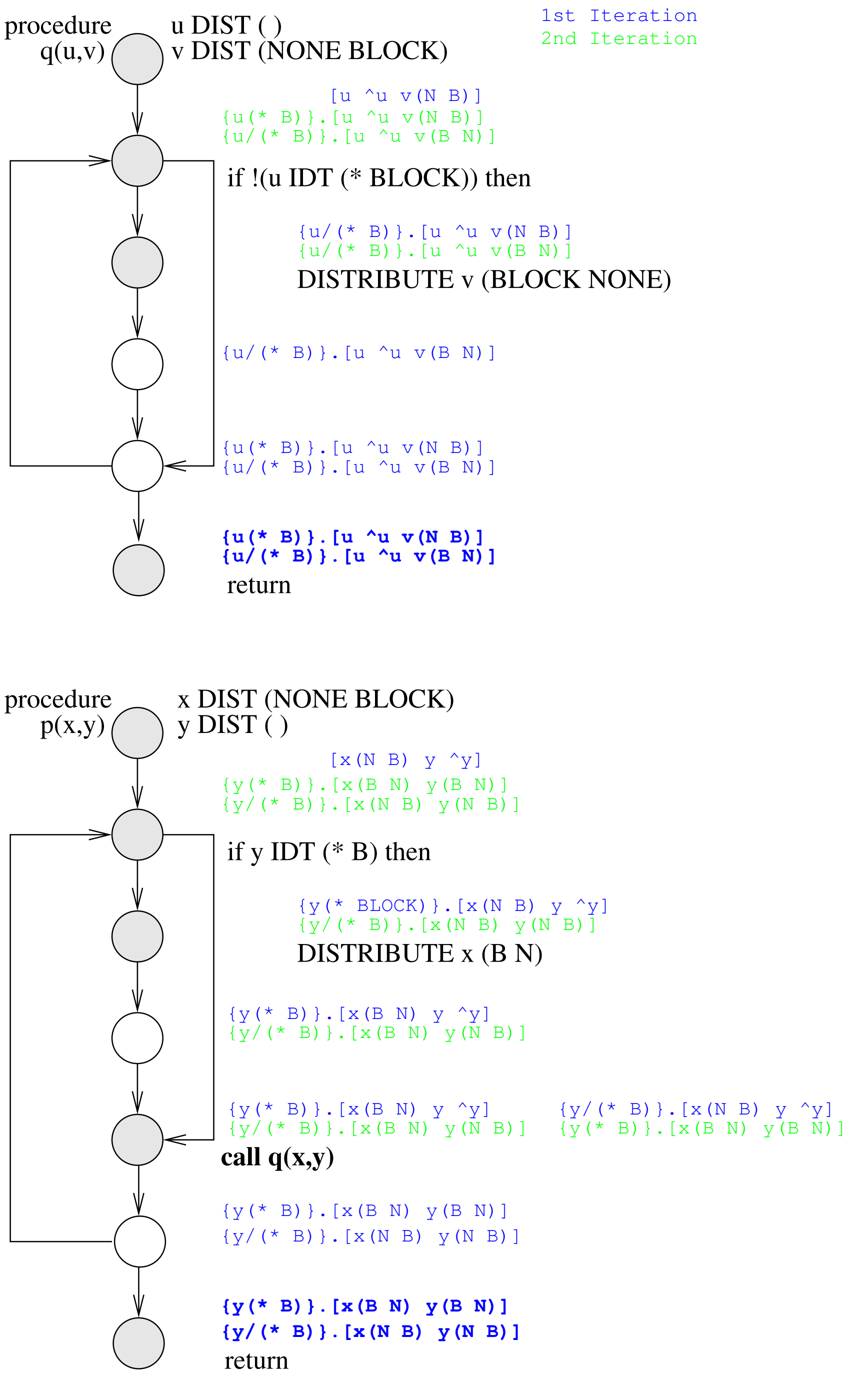


Fig. 2. Solution of MASKED_DISTRIBUTION_STATES for the procedures q and p (pass 1)

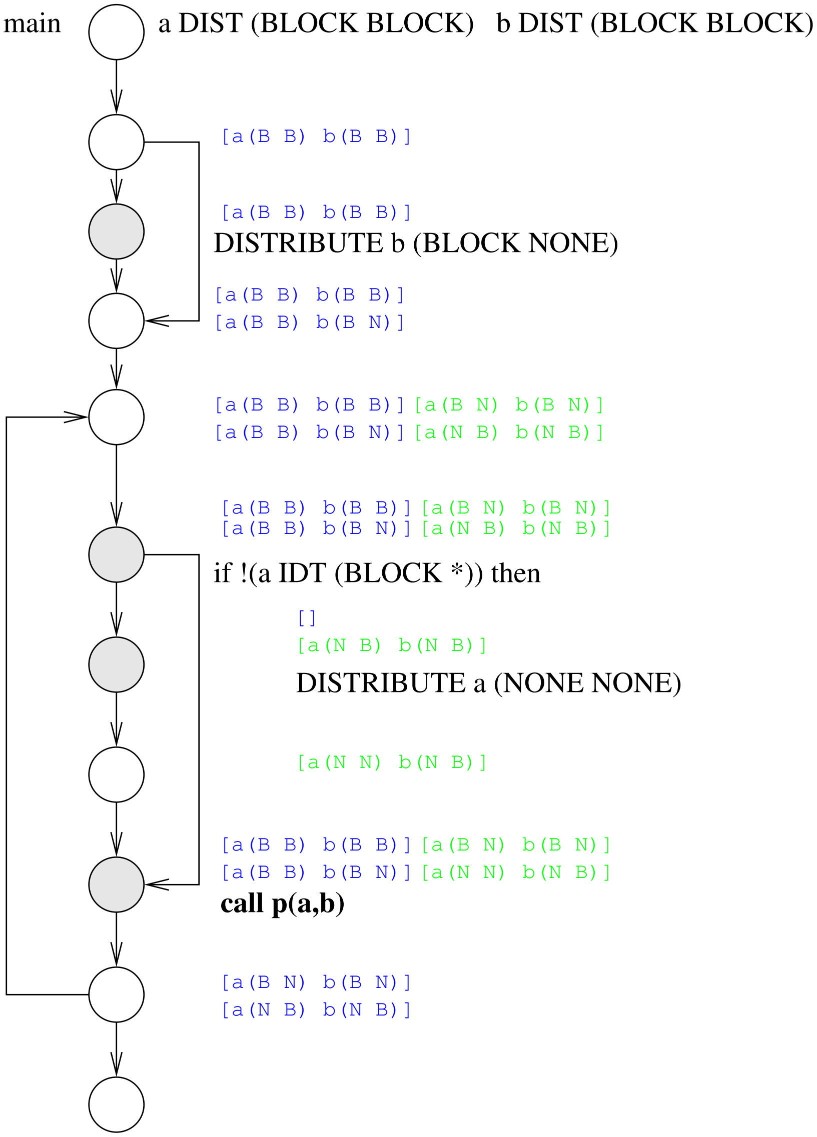

main
a DIST (BLOCK BLOCK)   b DIST (BLOCK BLOCK)
[a(B B) b(B B)]
[a(B B) b(B B)]
DISTRIBUTE b (BLOCK NONE)
[a(B B) b(B B)]
[a(B B) b(B N)]
[a(B B) b(B B)] [a(B N) b(B N)]
[a(B B) b(B N)] [a(N B) b(N B)]
[a(B B) b(B B)] [a(B N) b(B N)]
[a(B B) b(B N)] [a(N B) b(N B)]
if !(a IDT (BLOCK *)) then
[]
[a(N B) b(N B)]
DISTRIBUTE a (NONE NONE)
[a(N N) b(N B)]
[a(B B) b(B B)] [a(B N) b(B N)]
[a(B B) b(B N)] [a(N N) b(N B)]
call p(a,b)
[a(B N) b(B N)]
[a(N B) b(N B)]

Fig. 3. Solution for the main procedure

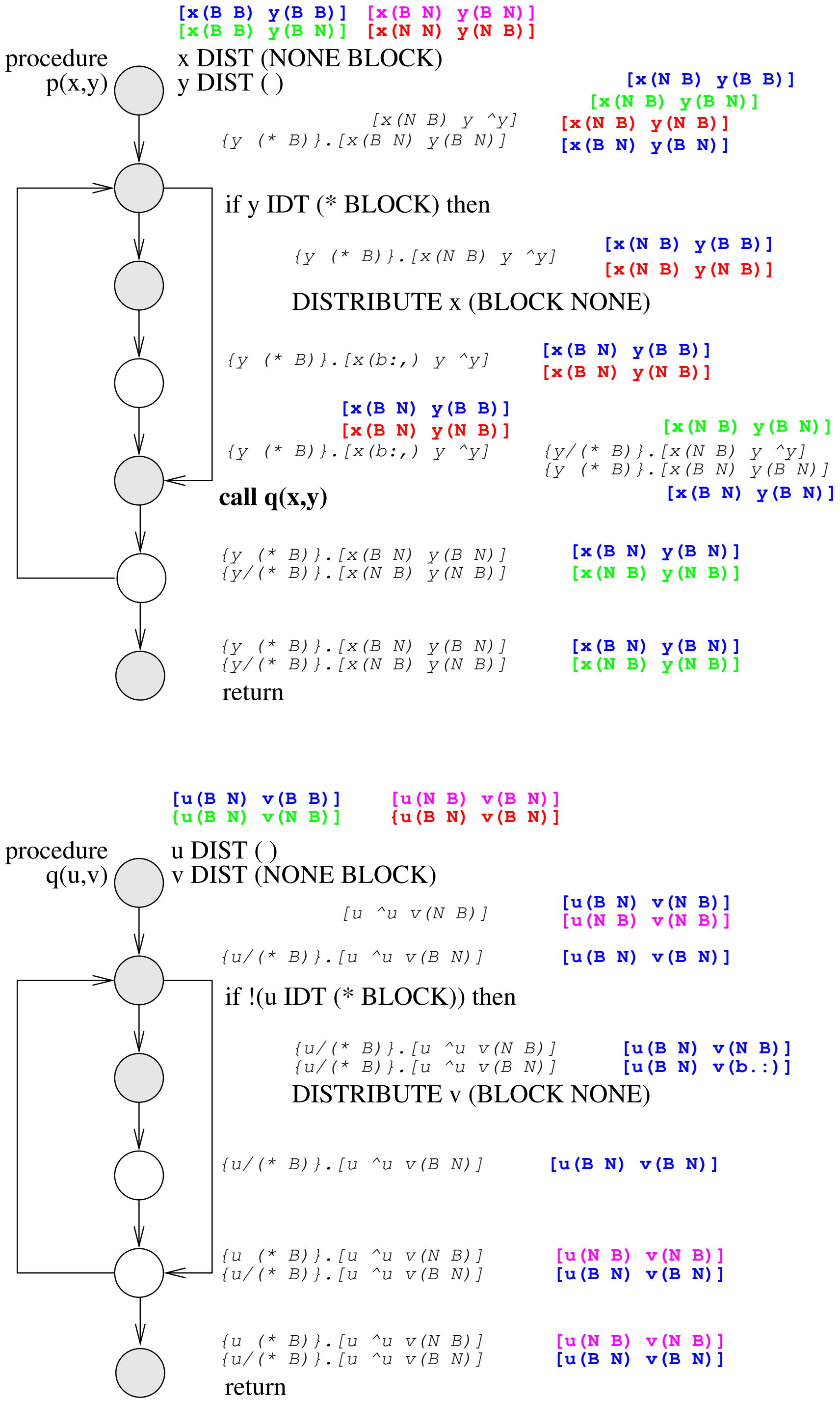


Fig. 4. Result for p and q after the down-propagation of states (pass 2)

Monotone bedingte Verzweigungen in Logikprogrammen

Ulrich Neumerkel und Stefan Kral

[1] TU Wien
ulrich@complang.tuwien.ac.at
[2] Fachhochschule Wiener Neustadt
stefan.kral@fhwn.ac.at

Zusammenfassung. Einfache bedingte Verzweigungen sind in den meisten Programmiersprachen ein elementares Sprachmittel. In monotonen Logikprogrammen war ihre Verwendung bisher auf nur sehr wenige Bereiche beschränkt, was oft zu unnatürlich komplexen oder inkorrekten Formulierungen führte. Wir stellen ein monotones Kontrollkonstrukt `if_/3` vor, das in vielen Fällen kompakte, korrekte und effiziente Definitionen direkt in ISO-Prolog erlaubt. Dies erreichen wir durch die explizite Darstellung von Wahrheitswerten mit Prädikaten höherer Ordnung. Zudem können auch Constraints unmittelbar einbezogen werden. Insbesondere werden dadurch Programme, die syntaktische Ungleichheit (`dif/2`) verwenden, wesentlich vereinfacht.

1 Einführung

Das Programmieren in purem, monotonem Prolog ist seit jeher ein wenig beachteter Bereich der Logikprogrammierung. Die meisten Prologprogramme bestehen noch immer aus unnötigerweise prozeduralen Elementen. Mit ein Grund dieses Missstandes sind oft Effizienzüberlegungen, die jegliche deklarative Sichtweise vernebeln. Um für pures Prolog eine brauchbare Programmiermethodik zu entwickeln, benötigen wir Konstrukte, die punkto Effizienz mit den nichtdeklarativen Methodiken vergleichbar sind. Wir richten dabei unsere Aufmerksamkeit ausschließlich auf monotone Elemente, die in vielerlei Hinsicht Vorteile bieten. So können alternative Beweisverfahren, wie etwa *iterative deepening* auf monotone Programme unmittelbar angewendet werden. Ähnlich verhält es sich mit deklarativen Debugging-Techniken und Program Slicing [5,10]. Ebenso wird der Constraintprogrammierung dadurch ein pureres Umfeld bereitet. Unsere Bemühungen zielen in eine ähnliche Richtung, wie es die Funktionale Programmierung erfolgreich vorgeführt hat: Fort von einer befehlsorientierten Sichtweise hin zum eigentlichen puren Kern des Paradigmas.

Viele Entwicklungen der letzten Jahre haben dazu beigetragen einen deklarativeren Programmierstil zu fördern. Ein besonders großer Fortschritt war die Klärung von Prologs ISO-Norm [8] durch Cor.2:2012 [13], das Laufzeitfehler semantisch abgeglichen hat [12] und das insbesondere die Programmierung Höherer

Ordnung über `call/N` auf feste, normative Beine stellte und damit weitergehende Verwendungen erlaubte [11]. Während das Konstrukt 1982 in ersten vagen Formen sondiert wurde [4] und bereits 1984 zum ersten Mal vorgeschlagen wurde [6], benötigte die präzise Definition offenbar ihre Zeit, um allgemein anerkannt und kodifiziert zu werden. Leider sind jedoch viele nichtdeklarative Konstrukte derzeit noch alternativlos. Ihre Verwendung für deklarative Zwecke ist zwar prinzipiell möglich, aber der Aufwand um dabei noch im puren Bereich zu verbleiben ist so hoch, dass er kaum in Kauf genommen wird.

Wir betrachten dazu zunächst die Schwächen von Prologs if-then-else Konstrukt und wenden uns dann jenen des Prädikats `member/2` zu, um daraufhin verbesserte Fassungen dieses Prädikats vorzustellen, die zur Einführung von Reifikation und letztlich eines monotonen if-then-else führen. Zu guter Letzt betrachten wir einige Beispiele zur allgemeinen Reifikation, die vor allem die Unterschiede zur konstruktiven Negation hervorheben.

2 Die Grenzen von Prologs if-then-else

Prologs if-then-else Konstrukt wurde erstmals um 1978 vom Interpreter des DEC10 Prologsystems implementiert [3], es wurde aber nicht durch den Compiler unterstützt. Erst spätere Systeme wie etwa Quintus Prolog verfügten über eine effiziente Implementierung, die schließlich zur Aufnahme in die ISO-Norm führte.

Wir betrachten lediglich Ziele der Form `( If_0 -> Then_0 ; Else_0 )`, wobei alle Teile einfache Ziele vor- oder benutzerdefinierter Prädikate sind. Hier ist die Ausführung äquivalent zu `( once(If_0) -> Then_0 ; Else_0 )`. Es wird also die erste Antwort von `If_0` genommen, weitere Antworten werden ignoriert. Weiters kommt `Else_0` nur dann in Betracht, wenn `If_0` scheitert. Das Konstrukt ist geeignet, um Prologs Negation zu implementieren, andererseits bedeutet das auch, dass zumindest dieselben Probleme auftreten: Nichtkommutativität der Konjunktion sowie Nichtmonotonie.

Weitere Verbesserungsversuche sind das sogenannte *soft cut*, das alle Antworten von `If_0` betrachtet. Es wird von einigen Systemen als `if/3` oder `(*->)/2` angeboten. Dadurch wird zwar die willkürliche Beschränkung auf die erste Antwort aufgehoben, und damit die Menge an Lösungen vollständiger und Antworten unabhängig von der Reihenfolge der Klauseln — modulo Termination und Fehler. Die eigentlichen, tieferliegenden Probleme der Nichtkommutativität und Nichtmonotonie bleiben jedoch bestehen.

Um die bisherigen Konstrukte sicher verwenden zu können, kommen für `If_0` nur Ziele in Betracht, die selbst zusichern, dass sie hinreichend instanziert sind. So etwa Prologs Arithmetikprädikate, die in Arithmetikausdrücken keine Variable zulassen, indem sie Instanzierungsfehler melden. Es gibt aber nicht viele weitere Prädikate, die sich in ähnlich sicherer Weise verwenden lassen. Insbesondere kommen Constraints dafür nicht in Frage. Darüber hinausgehende Ziele für `If_0` können nur in speziellen, kaum dokumentierten und ungeprüften Modi verwendet werden.

3 Die Schwächen von member/2

Auch an sich pure Definitionen weisen problematische Eigenschaften auf. Wir erläutern dies anhand von member/2, das für ein Ziel member(X, Es) wahr ist, wenn X Element der Liste Es ist.

```
member(X, [X|_Es]).
member(X, [_E|Es]) :-
   member(X, Es).
```

Die bekannte Relation ist etwas zu allgemein gefasst, da sie auch für Nicht-Listen erfüllt ist, die einen Listenpräfix mit dem passenden Element besitzen. So gilt etwa member(a, [a|non_list]). Derartige Verallgemeinerungen werden in Prolog jedoch gern in Kauf genommen, weil man sich dadurch eine effizientere Ausführung erwartet. Für die erste Antwort muss nur der Anfang der Liste bis zum ersten passenden Element betrachtet werden. Allerdings wird bei Wiedererfüllung dann doch noch die gesamte Liste betrachtet. Man bleibt also trotz Verallgemeinerung auf den Kosten zum Besuch der gesamten Liste sitzen. Dies ist einfach schon dadurch begründet, dass die verbleibende Liste ja tatsächlich noch ein weiteres passendes Element besitzen könnte.

```
?- member(1, [1,2,3,4,5]).      ?- member(1, [1,2,1,4,5]).
   true                            true
;  false.                       ;  true
                                ;  false.
```

Ein Ziel member/2 wird praktisch nie deterministisch sein und wird für die gesamte Dauer des Beweises Platz benötigen. Es ist naheliegend sich in dieser Situation nicht-deklarativer Hilfsmittel zu bedienen. Man beschränkt sich etwa auf die erste Antwort — ungeachtet der dadurch bedingten Unvollständigkeit. Ein weit verbreitetes Bibliotheksprädikat dazu ist memberchk/2.

```
memberchk(X, Es) :-      ?- X = 2, memberchk(X, [1,2]), X = 2.
   once(member(X, Es)).     X = 2.

                         ?-        memberchk(X, [1,2]), X = 2.
                            false.    % unerwartetes Scheitern
```

Offenbar wird so nicht nur die Monotonieeigenschaft verletzt, es gibt nun überhaupt keine deklarative Erklärungsmöglichkeit mehr. Als einzige Erklärung verbleibt das schrittweise Nachvollziehen des prozeduralen Ablaufs. Häufig wird diese Unzulänglichkeit kaschiert, indem man nur memberchk/2-Ziele zulässig erklärt, die hinreichend instanziert (*sufficiently instantiated*) sind, ohne sich allerdings auf eine genaue Definition dieses Kriteriums festzulegen und ohne Rückmeldung in Form eines Laufzeitfehlers. Man kann sich also nur bei variablenfreien Zielen sicher sein, dass memberchk/2 korrekt verwendet wird.

4 Ein grundüberholtes `member/2`

Das eigentliche Problem von `member/2` ist nicht sosehr Prologs Ausführungsmechanismus als die ursprüngliche Definition selbst, die bei einer gefundenen Antwort noch weitere redundante Antworten zulässt. Durch die folgende äquivalente, alternative Formulierung tritt die Ursache der redundanten Antwort, die durch die erste subsumiert wird, etwas deutlicher zutage. Die erste Alternative `X = E` wird in der zweiten nicht ausgeschlossen. Es ist also gut möglich, dass `X = E` auch für die zweite Alternative gilt. Genau an dieser Stelle muss also die Ungültigkeit von `X = E` und damit die syntaktische Ungleichheit der Terme zugesichert werden. In der neuen Definition `memberd/2` verwenden wir dazu das zu Anbeginn in Prolog 0 [1] vorhandene und leider ab Prolog I [2] für lange Zeit vergessene Prädikat `dif/2`, welches bisher nicht in die ISO-Norm von Prolog aufgenommen wurde. Für unsere Zwecke ist es unerheblich, ob `dif/2` wie vorgesehen Constraints verwendet, oder durch ISO-konforme Instanzierungsfehler falsche Antworten meidet. Wird `dif/2` nicht als implementierungsspezifische Erweiterung der ISO-Norm [8] bereitgestellt, genügt die Definition im Anhang.

```
member(X, [E|Es]) :-           memberd(X, [E|Es]) :-
   (  X = E                       (  X = E
   ;  member(X, Es)               ;  dif(X, E),
   ).                                memberd(X, Es)
                                  ).

?- member(1, [1,X]).           ?- memberd(1, [1,X]).
   true                           true
;  X = 1. % redundante Antwort ;  false. % nichtdet. Scheitern

                               ?- memberd(1, [1,2,3]).
                                  true
                               ;  false. % nichtdet. Scheitern
```

In dieser Formulierung von `memberd/2` treten nun keine redundanten Antworten mehr auf, dennoch verbleiben überflüssige Wahlpunkte, die **nichtdeterministisches Scheitern** verursachen und durch `; false` angezeigt werden. Dieses Problem tritt oft auf, wenn man versucht, mittels `dif/2` pure Programme zu definieren.

5 Reifizierung der Gleichheit

Das Problem dahinter liegt hier darin, dass durch unterschiedliche Alternativen miteinander zusammenhängende Fälle beschrieben werden. In einer Alternative ist Gleichheit und in der anderen Ungleichheit derselben Terme beschrieben. Durch Komprimierung dieser beiden Fälle in ein einziges Prädikat `(=)/3` hat nun eine Implementierung die Freiheit, Wahlpunkte, so möglich, zu vermeiden. Im Anhang findet sich dazu eine normkonforme Definition, die bereits viele überflüssige Wahlpunkte sofort entfernt.

```
memberd(X, [E|Es]) :-               =(X, X, true).
   =(X, E, T),                      =(X, Y, false) :-
   (  T = true                         dif(X, Y).
   ;  T = false,
      memberd(X, Es)
   ).
```

Diese Implementierung ist noch etwas verbesserungswürdig. Einige Prologsysteme sind nicht in der Lage die Disjunktion effizient zu implementieren. Zudem sollten auch fehlerhafte Werte für T erkannt werden. Weiters ist die Verwendung der Hilfsvariable T zur Darstellung des Wahrheitswertes besonders fehleranfällig. All diese Probleme werden wir durch einen neuen Ansatz lösen.

6 Das monotone if_/3

Die angeführten Probleme lassen sich allesamt durch ein neues Prädikat if_/3 beheben: if_(If_1, Then_0, Else_0). Die Bedingung wird nun nicht mehr durch ein einfaches Ziel dargestellt, sondern durch ein partielles Ziel eines reifizierten Prädikats, dem noch ein weiteres Argument zum vollständigen Ziel fehlt. Auf diese Art wird die Hilfsvariable für den Wahrheitswert versteckt. Damit gelangen wir nun zur endgültigen Fassung von memberd/2:

```
memberd(X, [E|Es]) :-               ?- memberd(1, [1,X]).
   if_( X = E                          true.
      , true
      , memberd(X, Es)              ?- memberd(1, [1,2,3]).
      ).                               true.
```

Diese Fassung vermeidet bereits überflüssige Wahlpunkte. Weitere Optimierungen sind durch partielle Auswertung möglich, um sämtliche Metacalls in if_/3 durch Ziele zu ersetzen.

Die Wahl, ein dreistelliges Prädikat zu verwenden und nicht mehrere binäre Operatoren, wie etwa bei Prologs if-then-else, war vor allem den semantisch sehr problematischen Nebeneffekten von if-then-else in ISO-Prolog geschuldet. So überschneidet sich das Konstrukt mit der Disjunktion, beide besitzen denselben äußeren Funktor (;)/2. Weiters ist (->)/2 für sich ein eigenes Konstrukt und führt damit zu einer sehr fragilen Semantik, die oft vom genauen Zeitpunkt abhängt, wann ein Term ein if-then-else Konstrukt beschreibt. Bei einem dreistelligen Prädikat können diese Probleme nicht auftreten. Dafür sind mehrfache Verzweigungen eher umständlich zu schachteln.

Aufbauend auf if_/3 lassen sich nun viele der üblichen Prädikate höherer Ordnung definieren - wobei sich nun wesentlich allgemeinere Verwendungen anbieten.

```
?- tfilter(=(X), [1,2,2], Fs).
   X = 1, Fs = [1]
;  X = 2, Fs = [2, 2]
;  Fs = [], dif(X, 2), dif(X, 1).

duplicate(X, Xs) :-              ?- duplicate(X, [1,2,2,1,3]).
   tfilter(=(X), Xs, [_,_|_]).      X = 1
                                 ;  X = 2
                                 ;  false.
```

7 Allgemeine Reifizierung

Bisher haben wir lediglich ein einziges reifiziertes Prädikat verwendet — für syntaktische Gleichheit. Für jede weitere neue Bedingung benötigen wir eine eigene Definition. Es obliegt also dem Programmierer, eine entsprechend reifizierte Fassung eines Prädikats zu erarbeiten. Damit unterscheiden wir uns grundsätzlich von allgemeineren Verfahren, wie etwa der Konstruktiven Negation [7,9], die mehr oder minder automatisch versucht, die Negation eines Ziels zu bilden. Ein Ansatz [7] hängt von richtig gesetzten delay-Deklarationen ab. Kann also keine unendliche Anzahl von Antworten erzeugen für Prädikate, die als Bedingungen gelten. Jedenfalls sind derartige Techniken verwendbar, um reifizierte Prädikate automatisch zu erzeugen. Effizient werden sie vermutlich jedoch nicht sein, da für den negativen Teil der positive nochmals in einem eigenen, vermutlich metainterpretierten Ausführungsmodus behandelt wird und da die Gemeinsamkeiten zwischen positivem und negativem Fall nicht geteilt werden können; zumindest also dieser Teil redundant ist. Wir werden dies nun anhand der Reifikation von `memberd/2` erörtern.

```
memberd_t(X, Es, true) :-
   memberd(X, Es).
memberd_t(X, Es, false) :-
   maplist(dif(X), Es).
```

Die Definition besteht für den negativen Fall darauf, dass `Es` eine wohlgeformte Liste ist, deren Elemente alle von `X` verschieden sind. Dies zeigt schon einen klaren Unterschied zur Konstruktiven Negation. Während in unserer Definition `memberd_t(X, non_list, T)` einfach nur scheitert, also weder wahr noch falsch ist, müsste diese bei konstruktiver Negation mit `T = false` erfolgreich sein. Einfach, weil `memberd(X, non_list)` scheitert, muss die Negation also gelten. Hingegen verlangt unsere Definition bis zu einem gewissen Grad einen Listentyp. Es gibt auch Fälle, in denen unsere Definition für Nicht-Listen erfüllt ist. Etwa gilt `memberd_t(1, [1|non_list], true)`. Die genaue Entscheidung, welcher Fall einem Wahrheitswert zugeordnet wird und welcher nicht, lässt sich nur aus rein implementierungstechnischen Erwägungen ersehen. Weder Typsysteme noch konstruktive Negation lassen ein solches Ermessen zu. Es ist diese Freiheit, die uns eine sehr effiziente Implementierung gestattet. Unmittelbar ist unsere Definition nicht sonderlich effizient, vor allem weil beide Klauseln völlig unabhängig voneinander sind, und zur Erzeugung von überflüssigen Wahlpunkten führen. Allerdings können wir nun die Gemeinsamkeiten beider Alternativen herausheben: Beide Alternativen bestehen zumindest auf einer gemeinsamen Präfixliste, welche an Stellen endet, an denen das gesuchte Element X vorkommt.

```
memberd_t(X, Es, T) :-        l_memberd_t([], _, false).
   l_memberd_t(Es, X, T).     l_memberd_t([E|Es], X, T) :-
                                 if_( X = E
                                    , T = true
                                    , l_memberd_t(Es, X, T) ).

firstduplicate(X, [E|Es]) :-  ?- firstduplicate(1, [1,2,3,1]).
   if_( memberd_t(E, Es)         true.
      , X = E
      , firstduplicate(X, Es) ?- firstduplicate(X, [1,2,2,1,3]).
      ).                         X = 1.

?- firstduplicate(X, [A,B,C]).
   X = A, A = B
;  X = A, A = C, dif(C, B)
;  X = B, B = C, dif(A, C), dif(A, C)
;  false.
```

Komplexere Strukturen erschweren die Identifikation des gemeinsamen Teils. Bei einer linearen Liste kann ja nur der gemeinsame Teil aus einem Präfix bestehen, damit gibt es praktisch keine Freiheiten. Bereits ein binärer Baum gibt viel weniger vor, da jene Zweige, die nicht das betrachtete Element enthalten nun je nach Implementierung von Relevanz sind oder nicht. Die erste Fassung fordert hier nur das Minimum, während die verbesserte Fassung die Relation etwas einschränkt.

```
treemember(E, t(E,_,_)).       tree_non_member(_, nil).
treemember(E, t(_,L,R)) :-     tree_non_member(E, t(F,L,R)) :-
   (  treemember(E, L)            dif(E, F),
   ;  treemember(E, R)            tree_non_member(E, L),
   ).                             tree_non_member(E, R).

treemember_t(E, Tr, true) :-
   treemember(E, Tr).
treemember_t(E, Tr, false) :-
   tree_non_member(E, Tr).

?- treememberd_t(2, t(1,non_tree,t(2,non_tree,non_tree)), T).
   T = true.
```

In der folgenden verbesserten Implementierung verwenden wir bereits die reifizierte Disjunktion. Während Disjunktion in purem, monotonem Prolog modulo Nichttermination und Laufzeitfehler kommutativ ist, gilt die Kommutativität der reifizierten Disjunktion nur bedingt.

```
treememberd_t(_, nil, false).
treememberd_t(E, t(F,L,R), T) :-
   call(
     (  E = F
     ;  treememberd_t(E, L)
     ;  treememberd_t(E, R)
     ),
   T).

?- treememberd_t(2, t(1,non_tree,t(2,non_tree,non_tree)), T).
   false.                                  % Einschränkung
?- treememberd_t(2, t(1,     nil,t(2,non_tree,non_tree)), T).
   T = true
;  false.
```

8 Schluss

Wir haben einen neuen, besonders einfachen Ansatz zur monotonen, bedingten Verzweigung vorgestellt, der bereits in seiner ersten Implementierung kostspielige Wahlpunkte effektiv vermeidet. Alle verwendeten Mittel sind zwar schon seit langem bekannt, die konkrete Zusammenstellung gab es bislang jedoch nicht.

Die vorgestellten Programme wurden in den letzten Jahren von den Autoren auf comp.lang.prolog und stackoverflow.com schrittweise entwickelt. Die wesentlichen Eckpunkte waren:

2009-10-15 ISO-dif/2 comp.lang.prolog
2012-12-01 Reification of term equality stackoverflow.com/q/13664870
2014-02-23 memberd/2 stackoverflow.com/a/21971885
2014-02-23 tfilter/3 stackoverflow.com/a/22053194
2014-12-09 if_/3 stackoverflow.com/a/27358600

Weitere Beispiele unter stackoverflow.com/search?q=[prolog]+if_

Literatur

1. A. Colmerauer, H. Kanoui, Ph. Roussel, R. Pasero. Un système de communication homme-machine en Français, Rapport de recherche, CRI 72-18. U.E.R de Luminy. Université d'Aix-Marseille. 1972-1973.
2. Ph. Roussel, Prolog, manuel de référence et d'utilisation. Groupe d'Intelligence Artificielle de Marseille-Luminy, 1975.
3. L. M. Pereira, F. C. N. Pereira, D. H. D. Warren. User's Guide to DECsystem-10 Prolog. 1978.
4. D. H. D. Warren. Higher-Order Extensions to Prolog - Are They Needed?, Machine Intelligence 10. 1982. Originally: International Machine Intelligence Workshop, Cleveland, April 1981, DAI Research Paper 154.
5. M. Weiser. Programmers Use Slices When Debugging. CACM 25(7): 446-452, 1982.
6. R. O'Keefe. Draft Proposed Standard for Prolog Evaluable Predicates. 1984. Kopie unter: http://www.complang.tuwien.ac.at/ulrich/iso-prolog/okeefe.txt
7. D. Chan. An Extension of Constructive Negation and its Application in Coroutining. NACLP 1989.
8. ISO/IEC 13211-1:1995 Programming languages - Prolog - Part 1: General core.
9. W. Drabent. What is failure? An approach to constructive negation. Acta Informatica 32(1):27-59, 1995.
10. U. Neumerkel, St. Kral. Declarative program development in Prolog with GUPU. 12th Workshop on Logic Programming Environments (WLPE), Copenhagen 2002.
11. U. Neumerkel. Lambdas und Schleifen in monotonen Logikprogrammen. KPS 2009.
12. U. Neumerkel, M. Triska. An error class for unexpected instantiations. ISO/IEC JTC1 SC22 WG17 N226. 2010. http://www.complang.tuwien.ac.at/ulrich/iso-prolog/error_k
13. ISO/IEC 13211-1:1995/Cor 2:2012. Second Technical Corrigendum for Programming languages - Prolog - Part 1: General core.

Anhang

Alle folgenden Definitionen sind strikt normkonform und benötigen daher keinerlei Constraintmechanismus. Sie finden sich in `library(reif)` für SICStus Prolog und andere Prologsysteme.

Die Definition von `dif/2` ist nur dann erforderlich, wenn ein Prologsystem keine eigene Implementierung von `dif/2` als implementierungsspezifische Erweiterung aufweist. Alle Fälle, in denen keine korrekte Antwort möglich ist, der Constraintmechanismus also benötigt werden würde, werden durch einen Laufzeitfehler angezeigt. Inkorrekte Antworten werden somit vermieden.
`if_/3` besteht auf einen konkreten Wahrheitswert für `T` in `call(If_1, T)`. Im Falle von Nichtdeterminismus sollte `If_1` zuerst `true` als Antwort liefern.
`(=)/3` ist bereits völlig deterministisch in allen Fällen, die durch syntaktische Gleichheit und Ungleichheit eindeutig sind. Möglicherweise ist es auch hier interessant, den Wahrheitswert mit dem Fehler `type_error(boolean, T)` abzusichern. Eine effizientere interne Implementierung könnte die bis zu vier Traversierungen der beiden Terme auf bestenfalls eine reduzieren.
`(',')/3` und `(;)/3` sind reifizierte Fassungen der Konjunktion und Disjunktion.

```
dif(X, Y) :-
   X \== Y,
   ( X \= Y -> true ; throw(error(instantiation_error,_)) ).

% :- meta_predicate(if_(1, 0, 0)).
if_(If_1, Then_0, Else_0) :-
   call(If_1, T),
   (  T == true -> call(Then_0)
   ;  T == false -> call(Else_0)
   ;  nonvar(T) -> throw(error(type_error(boolean,T),_))
   ;  /* var(T) */ throw(error(instantiation_error,_))
   ).

=(X, Y, T) :-
   (  X == Y -> T = true
   ;  X \= Y -> T = false
   ;  T = true, X = Y
   ;  T = false,
      dif(X, Y)                              % ISO extension
      % throw(error(instantiation_error,_)) % ISO strict
   ).

','(A_1, B_1, T) :-
   if_(A_1, call(B_1, T), T = false).

;(A_1, B_1, T) :-
   if_(A_1, T = true, call(B_1, T)).
```

Obstacles to Compilation of Rebol Programs

Viktor Pavlu

TU Wien
Institute of Computer Aided Automation, Computer Vision Lab
A-1040 Vienna, Favoritenstr. 9/183-2, Austria
`pavlu@caa.tuwien.ac.at`

Abstract. Rebol's syntax has no explicit delimiters around function arguments; all values in Rebol are first-class; Rebol uses *fexprs* as means of dynamic syntactic abstraction; Rebol thus combines the advantages of syntactic abstraction and a common language concept for both meta-program and object-program. All of the above are convenient attributes from a programmer's point of view, yet at the same time pose severe challenges when striving to compile Rebol into reasonable code. An approach to compiling Rebol code that is still in its infancy is sketched, expected outcomes are presented.

Keywords: first-class macros, dynamic syntactic abstraction, $vau calculus, fexpr, Kernel, Rebol

1 Introduction

A meta-program is a program that can analyze (read), transform (read/write), or generate (write) another program, called the object-program.

Static techniques for syntactic abstraction (macro systems, preprocessors) resolve all abstraction during compilation, so the expansion of abstractions incurs no cost at run-time. Static techniques are, however, conceptionally burdensome as they lead to staged systems with phases that are isolated from each other. In systems where different syntax is used for the phases (e. g., C++), it results in a multitude of programming languages following different sets of rules.

In systems where the same syntax is shared between phases (e. g., Lisp), the separation is even more unnatural: two syntactically largely identical-looking pieces of code cannot interact with each other as they are assigned to different execution stages.

While static approaches to syntactic abstraction try to alleviate the burden on the programmer by lowering the barriers between phases (e. g., `constexprs` introduced in C++11 can be used at compile-time and in the compiled program) we are interested in making conceptually simple dynamic abstractions more efficient.

Recent work [7] has shown that fexprs can in fact be used to bring the two phases together in a single dynamic syntactic abstraction system.

Vau expressions as defined by Shutt create operative combiners (operatives) based on statically scoped fexprs. An operative combiner is a function that does not evaluate its operands before application. Instead, operatives work directly with their operands, not on their argument values. Operands are passed unevaluated together with access to the calling environment, so that the operands may be evaluated explicitly when needed by the operative. This shifts from an implicit-evaluation environment to an environment where the evaluation of operands is controlled explicitly.

Operative combiners have access to both the static (or lexical) environment where the combiner was created and the dynamic environment where the combiner is applied. When using the dynamic environment to evaluate the body of the combiner, we get dynamic scoping as in LISP. When using the static environment to evaluate the body of the combiner, we get static scoping as in Scheme. Both can be implemented using Shutt's operative combiners as the vau abstraction provides explicit access to both environments.

The shift from an implicit-evaluation environment to an environment where evaluation of operands is explicitly requested avoids frequent difficulties with (naive) macro implementations. Consider the running example in Figs. 1–4. `or` shall be defined with short-circuit evaluation, so that it returns the value of its first operand if it evaluates to true and otherwise returns the result of evaluating the second operand.

A naive approach to define `or` as applicative (or lambda) is shown in Fig. 1. The problem with this definition is that the second operand will be evaluated already when `or` is applied, no matter what the value of the first operand is. The definition therefore lacks the required short-circuit evaluation.

```
>> (define (or x y)
     (if x
         x
         y))

>> (or 1 something-undefined)
reference to undefined identifier: something-undefined
```

Fig. 1. Flawed defintion of `or` as an applicative in Racket. Operands to an applicative are evaluated to argument values during application, so the required short-circuiting cannot be provided by applicatives.

Using pattern-based macros we can define `or` as a macro that rewrites to a conditional (cf. Fig. 2) and offers the required short-circuiting but the first operand is evaluated twice in the expanded code which in the least creates bloat but also causes unwanted results when the evaluation has side-effects.

By the use of a local variable to cache the value of the first operand, the macro can avoid duplicate evaluation (cf. Fig. 3). The introduction of a new local variable, however, introduces another set of problems with accidental name captures that hygienic macro implementations solve for the programmer [1]. Still, the macro definition is obfuscated with code that circumvents multiple evaluation as there is no means of referring to a parameter without triggering its evaluation in an implicit-evaluation environment.

In the context of explicit-evaluation operatives this is different: referring to a parameter and evaluation of a parameter are distinct. Shutt's Kernel programming language [6]

```
>> (define-syntax-rule (or x-exp y-exp)
     (if x-exp x-exp y-exp))

>> (or (print "first") (print "second"))
"first""first"
```

Fig. 2. Flawed defintion of a short-circuiting `or` as a pattern-based macro in Racket. The first operand is evaluated twice.

```
>> (define-syntax-rule (or x-exp y-exp)
     (let ([x x-exp])
       (if x x y-exp)))

>> (or ((get-print) "first") ((get-print) "second"))

>> (or (print "first") (print "second"))
```

Fig. 3. Definition of a short-circuiting `or` as a pattern-based macro in Racket. Caching the result of the evaluated first operand avoids multiple evaluation.

based on vau expressions thus allows for a straight-forward definition of the `or` macro as shown in Fig. 4. No code to circumvent multiple evaluation obfuscates the actual algorithm. The calls to `eval` make it explicit where evaluation is performed and within which environment.

```
(define-vau (or x-exp y-exp env) env
  (let ([x (eval x-exp env)])
    (if x x (eval y-exp env))))
```

Fig. 4. Definition of a short-circuiting `or` operative in Kernel (with Racket Syntax).

Operatives can choose to implement any evaluation strategy and may even choose not to evaluate but analyze its operands. An operative may then compose code based on the syntactic structure of its operands, which is what macro systems offer, only at run-time. As a result, the core language is drastically simplified, as many language features usually built-in to the language can be constructed from vau expressions, e. g., Macros, special forms, applicative and operative combiners can all be constructed from vau expressions. Applicative combiners or lambdas, i. e., functions that evaluate operands to arguments, are just operative combiners where all operands are evaluated to argument values.

Shutt has demonstrated that a language based on vau expressions can use a small axiomatic set of primitives for both the macro language and the target language.

Dynamic syntactic abstraction using vau expressions thus promises both, the general advantages of syntactic abstraction in e. g., crafting domain-specific (sub-)languages and a common language concept. The phase separation is overcome as syntactic abstraction is then an equal among the other abstraction mechanisms, increasing the use of syntactic abstraction and making it possible for interactive languages to have syntactic abstraction in the first place.

2 Problem

When operatives are primitive data types of the language (first-class operatives) the separation in a metaprogramming- and a programming stage is lifted from the language. What would otherwise be conceptually isolated may now freely interact, i. e., not only may the same language and primitives be used in the meta-program and the object program (homogenous metaprogramming), they may also share the same data as the phase separation is overcome.

This unique feature of dynamic syntactic abstraction, however, comes at a price: when operatives are deliberately indistinguisable from applicatives, it is generally no longer possible to determine whether a combiner is an applicative (works on operands evaluated to arguments) or an operative (works on unevaluated operands), until immediately before evaluation (when the operator of a combination is evaluated to an applicative or operative combiner). This is not an unwanted side-effect but the direct result of treating operative combiners like any other value in the language. Still, it presents a practical difficulty.

Expressions in a program with first-class operatives can no longer be grouped into expressions that will be evaluated to argument values and expressions that must remain unmodified. Hence, two expressions that evaluate to the same value are no longer interchangeable in any context because an operative combiner may distinguish between those expressions on purely syntactic grounds, e. g., `(print (add 3 1))` must not be replaced with the shorter `(print 4)` since `print` may or may not be an operative that distinguishes between combinations and literals.

In effect, all expressions must remain unmodified in order to not alter the meaning of the program as long as it is not known whether an expression is to be evaluated by an applicative or operative combiner. This precludes all program optimization since there is no way to statically distinguish between applicative and operative combiners in the general case.

2.1 The Kernel Programming Language

Kernel [6] is the programming language implemented by Shutt to demonstrate the practicability of fexpr-based dynamic syntactic abstractions.

Even without the transformation-adverse properties introduced by first-class operatives, Kernel is not an easy target for program analysis:

- We cannot statically determine the value of any variable; all combiners are first-class and kept in variables.
- We cannot statically compute the value of even the simplest arithmetic expressions (e. g., $1+1$) because mathematical operators are also combiners and may have been redefined to a non-standard binding.

- Non-standard bindings of combiners may also alter the number of operands the combiner uses, resulting in a different program structure.
- We cannot statically determine the list of free variables in a code block.

Kernel is thus implemented as a simple non-optimizing interpreter due to the properties of the language.

2.2 The Rebol Programming Language

Rebol [4] has the same concept of operands being passed unevaluated but calls them *get arguments* and the choice of evaluation or non-evaluation lies with each individual operand rather than with the type of combiner (applicative or operative). So while Kernel only allows purely applicative or purely operative combiners, Rebol programs may have mixed combiners where some values are passed as unevaluated operands and some as evaluated arguments.

Further, Rebol has no syntax to denote the list of operands in a combination but instead each combiner has a fixed number of operands known from the combiner's definition. Before applying a function, the exact number of operand expressions is evaluated and then passed as argument values.

With the potential for redefinitions to a different number of operands this is an obstacle for static analysis in its own right, that is orthogonal to the problem of discerning applicative and operative operands.

The example in Fig. 5 illustrates the practical implications of this. It shows how a programmer would interpret the application of three pre-defined functions in Rebol. Given this example, a Rebol programmer knows that `replace` consumes the three values and replaces all occurrences of "bar" with "baz" in the string "foo". The call to `append` returns a new string "foobar" and is followed by the (unconsumed) literal "baz", and the third function application using `print` is read as `(print "foo")` followed by two string literals. With the information on the number of arguments a function consumes, a program analysis would be able to interpret this program fragment in the same manner. If, however, the number of arguments a function consumes is not known statically, as is the case in the example given in Fig. 5 where the arity of function `f` depends on a value only known at runtime, neither programmer nor analysis are able to statically discern between operand values and further values that follow the function application.

While making the definition of a function's arity depend on some completely random value is clearly an artificial example, it is definitely of practical relevance to a language as dynamic as Rebol that the definition of a function depends on a value only known at runtime. This is always the case when an identifier is used to abstract over two or more implementations of a function, e. g., when a generic `open-db` function is used to establish the connection to a database, and depending on the particular database system used, one or the other `open-db`-implementation is assigned to `open-db`. Then, at least, the case of a change in arity is ruled out for practical reasons, although in theory is still possible.

The absence of syntax to delimit the operand list has an additional effect in Rebol. Applicatives evaluate their arbitrarily nested operand expressions to single argument values while operatives only consume their first operand unevaluated as single value

```
>> replace "foo" "bar" "baz"
;; read as: (replace "foo" "bar" "baz")

>> append "foo" "bar" "baz"
;; read as: (append "foo" "bar") ("baz")

>> print "foo" "bar" "baz"
;; read as: (print "foo") ("bar") ("baz")

>> switch random 3 [
     1 [f: :replace]
     2 [f: :append ]
     3 [f: :probe  ]
   ]

>> f "foo" "bar" "baz"
;; read as: ???
```

Fig. 5. Function application syntax in Rebol has no clue to the number of arguments a funtion will consume. Without the implicit information on the arity of functions, it is impossible to parse the tokens following the application of `f` into arguments and non-arguments.

with no attention to the arity of sub-expressions. The same code `f g x` where `f` and `g` are combiners with arity 1 can therefore result in different parse trees depending on the kind of combiner in `f`: with an applictive combiner, the code reads as `(f (g x))`, whereas an operative combiner results in `(f g) (x)`.

An example is given in Fig. 6. In the applicative combiner, the operand expression `add1 1` is reduced to `2` and passed as argument value. `42` remains as the next value to be consumed. Meanwhile, in the operative example, the operand expression consists of `add1` only and `1` remains as the next value to be consumed.

So Rebol, too, has very limited room for static optimization and is implemented as a simple interpreter.

3 Expected Results

By introducing a strict import/export module system to a language with dynamic syntactic abstraction, we essentially replace the implicit stage boundaries common to static syntactic abstraction techniques with explicit module boundaries. We expect three main results of this change.

3.1 Increased Flexibility Inside

We believe that the creation of domain-specific languages is a very powerful tool that is best used within a closely confined part of a program. Within a module there will be no separation between meta- and object program and no phase separation when using

```
>> add1: func [x][ x + 1 ]

>> applicative: func [x][ probe x ]
>> operative: func  [’x][ probe x ]

>> print-next: func [a b][
     print ["next value:" b]
   ]

>> print-next applicative add1 1 42
2
next-value: 42

>> print-next operative   add1 1 42
add1
next-value: 1
```

Fig. 6. Two combiners are evaluated. The applicative reduces a nested expression to a single value while the operative passes a single operand without reduction, resulting in different associations of operands.

first-class operatives. A single base language can be used to both extend the language and write programs using the base or extended languages.

Between modules, the flexibility is curtailed. Any modifications to the base language are restricted to module boundaries. We believe this to be the level of flexibility that is practical when using such a powerful tool. For small problems, domain-specific sub-languages have successfully been used as effective tools. For programming in the large, however, we do not believe that it is of advantage if all combiners, operators and functions as well as names can be redefined at a single place and all other, entirely unrelated places that use the same entity, are affected.

3.2 Modularization of Modifications

While stage boundaries prevent program phases from all interactions with each other (even wanted interactions of seemingly compatible parts), the module boundaries

- separate different program parts and prevent *accidental* interaction of distant program parts with each other,
- explicitly state the interface between separate program parts,
- document the scope of semantic abstractions,
- aid with selective imports of syntactic abstractions and thus encourage reuse of abstractions.

This simplifies the creation of domain-specific languages as accidental side-effects of language modifications are better isolated. Errors due to these side-effects are then easier to locate and debug.

All of which fosters locality and modularization of language extensions, therefore increasing clarity of a program with syntactic abstraction and adding to hygiene. This will add to the attractiveness of domain-specific languages and dynamic syntactic abstraction.

3.3 Room for Static Analysis

Explicity stated imports/exports between modules confine the flexibility of the self-modifying language within module boundaries to a degree where static analysis and optimization becomes useful again.

When the flexibility that code from one module leaks into another module is ruled out and the only interaction between modules is documented through the imports, we anticipate that modules are self-contained to the degree that all analysis is essentially whole-program analysis, so static analysis becomes useful again. The interplay with other modules need not be analyzed at all, as information on external objects is already available through the imports each module is instructed to follow.

From the division into libraries we expect that separate ahead-of-time compilation to machine code is feasible and allows efficient implementation of syntactic abstractions without stage boundaries.

4 Methods

Aim of this work is to research practical methods to achieve separate compilation of subexpressions at the module level in a language with first-class operatives as dynamic syntactic abstraction.

We will primarily focus our efforts on the Kernel programming language as several problems with Rebol are avoided there while the metaprogramming characteristics that make Rebol interesting are retained. In particular, Kernel has a clean definition of the language (not just a reference implementation), several implementations are available and it does not share the implicit arity of combiners found in Rebol, so we can concentrate on the problem of separate compilation of first-class operatives.

The first steps are:

- We analyze library systems that allow the selective importation and exportation of symbols for their suitability to a language based on vau expressions. As starting point we use the Scheme library system defined in R7RS small language [5].
- Leveraging the customized library system we add devices to the language that separate programs into independent modules with an explicitly defined interface. We expect that tying down all flexibility between modules to what is explicity declared in the imports/exports will allow us to conservatively treat each module as a complete program, enabling whole-program analysis at the module level.
- A static data-flow analysis will be formulated as whole-program analysis to find stable bindings of operatives and applicatives. With this information in place, safe optimizing transformations can be implemented.
- For pathological programs that abuse the flexibility of the language (i. e., repeated redefinition of core operatives that confuses static analysis) the analysis will produce

warnings as we think that those cases are rarely of practical use. Compilation of programs with such constructs must be treated in a convservative way, however. Fallback to interpretation at run-time is an option. The rationale here is that programs are analyzed and optimized where possible and interpreted at run-time where necessary.
- Shutt comments briefly on possible inlining transformations for hygienic and unhygienic fexprs. This seems a viable starting point for optimizations orthogonal to module-based optimization and approximate typing.

The benefits of Shutt's vau expressions can be evaluated by direct comparison of programs with first-class operatives to a similar program written in a language without this language construct. A more rigorous evaluation of vau expressions is difficult as the number of real-world programs in Kernel is very small. The motivation for vau expressions is, however, not the main focus of this work. Our goal is to investige restrictions on programs with vau expression that do not impede their usefulness as a means of syntactic abstraction and, at the same time, create room for static analysis and optimization.

The fact that first-class operatives are not encumbered by our restrictions will be validated by creating an implementation of Kernel under these restrictions. In Kernel, the operatives are not merely a device added on top of the languge to enhance programmer productivity (which, in addition, would be laborious to evaluate), but are the basic abstraction of the language from which all other abstractions are built, i. e., `$lambda`, the primitive to define functions, is an application of the operative `$vau`. Consequently, if it is possible to implement Kernel with our restrictions in place, the viability of the restrictions is demonstrated.

The other part to be validated is that the restrictions actually introduce a potential for analysis and optimization and to a lesser degree, that the optimizations are in fact beneficial to the efficiency of programs. The feasibility of optimizing ahead-of-time compilation will be established by implementing a system capable of modular compilation of a language with first-class operatives. This is an open question left for future work in Shutt's thesis [7] and the central point of this work.

Once the potential for optimization is in place, it is then interesting to evaluate the quality of optimizations in terms of fast execution. This is primarily a question of quality regarding the analyses and transformations but also regarding the potential for optimizations attained through restrictions on the flexibility of the language, which, again, is the center of interest in our work. Evaluation of performance will be done using benchmarks derived from the (few) existing Kernel and (more) Rebol programs. Roughly 1200 programs are available through the Rebol Script Library [3] that can be used for this purpose.

5 Related Work

Macros are the predominant form to implement syntactic abstraction. They range from simple token-based substitutions over pattern-based substitution systems (syntax-rules macros used in R5RS [2]) to meta-programs that can use arbitrary functions to create their object programs (syntax-case macros used in R6RS [1]). They have in common that they operate in an implicit-evaluation environment so that naively written macros may suffer from undesired multiple evaluation of macro operands. A way around this is to have the programmer manually ensure that macro parameters are at most evaluated once and the results be kept in a variable local to the macro expansion for use inside the expanded code. Hygienic macro expansion automatically takes care that local names of such variables do not inadvertedly capture bindings at the macro expansion site. Circumventing multiple evaluation obfuscates the actual algorithm but is necessary as there is no means of referring to a parameter without triggering its evaluation in an implicit-evaluation environment.

C++ Templates can be seen as a macro system that expands code to cater for different types. It can also be used to perform integer and pointer computations at compile-time.

All these syntactic abstraction mechanisms, being static techniques, share the same conceptual divide in a generating phase (macro expansion) and a phase where the generated code is executed. Sometimes the phases share the same language (e. g., Lisp) resulting in an even more unnatural separation as two syntactically largely identical-looking pieces of code cannot interact with each other because they are assigned to different execution stages.

Dynamic syntactic abstraction using operatives was pioneered by Shutt in his thesis [7]. His operatives are implemented using statically-scoped fexprs and work like normal functions (i. e., applicatives), except that the operand expressions are passed unevaluated.

Wand [8] demonstrated that the equational theory of fexprs is trivial which means that two expressions in the language can only be used interchangeably (are contextually equivalent) if they are syntactically identical (α-congruent). In essence, this observation precludes all optimizations as expressions cannot be replaced by anything except themselves without possibly altering the meaning of the program.

Shutt addresses this result and traces back the seeming contradiction with his thesis to differences in the modeled language. If there is any difference between two expressions that is observable by a fexpr, the two expressions are no longer contextually equivalent. In Wand's language, everything was an S-expression and could thus be deconstructed by fexprs. As a result, only S-expressions that were identical had contextual equivalence and the equational theory was indeed trivial. In Shutt's language, however, not every entity is a decomposable S-expression. There are encapsulated objects (environments, compound operatives) and computational states (active terms) which have a non-trivial equational theory and leave potential for optimizing transformations.

6 Conclusions

We believe that adapting a rigorous library system to an otherwise hardly restricted language is a viable first step to attain optimizing ahead-of-time compilation of programs with syntactic abstractions based on first-class operatives. Aim of this work is to demonstrate the feasibility of this approach.

References

1. Dybvig, R., Hieb, R., Bruggeman, C.: Syntactic abstraction in scheme. LISP and Symbolic Computation 5(4), 295–326 (1993), `http://dx.doi.org/10.1007/BF01806308`
2. Kohlbecker, E.E., Wand, M.: Macro-by-example: Deriving syntactic transformations from their specifications. In: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 77–84. POPL '87, ACM, New York, NY, USA (1987), `http://doi.acm.org/10.1145/41625.41632`
3. Rebol Special Interest Group: Rebol Script Library. `http://www.rebol.org/script-index.r`
4. Sassenrath, C.: Rebol design objectives for computer scientists. `http://www.rebol.com/docs/design-objectives.html`
5. Shinn, A., Cowan, J., Gleckler, A.A.: Revised^7 report on the algorithmic language Scheme. Tech. rep., Scheme Steering Committee (2013)
6. Shutt, J.N.: Revised^{-1} report on the kernel programming language. Tech. rep., Worcester Polytechnic Institute (2009)
7. Shutt, J.N.: Fexprs as the basis of Lisp function application or $vau: the ultimate abstraction. Ph.D. thesis, Worcester Polytechnic Institute (2010)
8. Wand, M.: The theory of fexprs is trivial. Lisp and Symbolic Computation 10(3), 189–199 (1998)

PyPy's Number Crunching Optimization

Richard Plangger and Andreas Krall

Institut für Computersprachen, Technische Universität Wien
planrichi@gmail.com, andi@complang.tuwien.ac.at

Abstract. PyPy is a widely known virtual machine for the Python programming language implemented in the RPython subset of Python which includes a tracing Just-In-Time (JIT) compiler. In this article the new auto vectorizer built into the RPython optimizing backend is presented. It uses the linear sequence of instructions of a trace to find parallelism. Dependency information is gathered and used to reschedule some of the instructions as vector statements.
This optimization is neither tailored for the NumPy library nor for a specific hardware architecture. Every interpreter written in RPython can benefit from the new optimization. To empirically evaluate the optimization, the x86 assembler backend has been extended to emit SSE4 vector instructions for the optimized traces. Preliminary results show that it is indeed possible to leverage the speed gain SIMD instruction sets offer. The implementation is not very complex and the optimizer is reasonably fast.

1 Introduction

PyPy is a widely known virtual machine for the Python programming language. Opposed to the standard implementation (CPython), it includes a tracing just-in-time (JIT) compiler. The implementation language is a statically typed subset of Python called Restricted Python (RPython). RPython is an abstraction for byte code interpreters and is able to automatically generate a garbage collector and a tracing JIT compiler. Thus it is not only used for PyPy but also for many other interpreters for dynamic functional languages or instruction set simulators.

In the last decade new Single Instruction Multiple Data (SIMD) instruction sets where built into processors to speed up multimedia applications. They are not only useful for multimedia applications but also for scientific applications. In theory, given a single precision floating point operation in a loop, if the loop is vectorized to SSE4 instructions (a x86 instruction set architecture (ISA) extension) it executes 4 times faster.

Recent developments in scientific computing have drawn attention to libraries for numerical computations (e.g NumPy). NumPy and others currently remove the interpreter overhead of numerical computations by writing the critical routine in a low level language. They are compiled to the host computers architecture ahead of time. At runtime the language interpreter invokes the foreign function compiled earlier. Since NumPy is a commonly used library, PyPy rewrote parts of NumPy and included it in the standard library. This setup renders most of the critical loops as normal program loops instead of foreign functions and makes it desirable to optimize such loops. To simplify optimizations arrays in NumPy are homogeneous, primitive typed and continuous in memory.

In this article the new auto vectorizer built into the RPython optimizing backend is presented. First details of PyPys tracing JIT compiler are presented, then the vectorizer is described, finally preliminary results on the performance are given.

2 Related Work

The building block for Tracing JIT compilation has been introduced in the Dynamo project [BDB00]. Dynamo is an transparent optimization system that operates on a binary executable. Interpretation starts to execute the program and observes backward branches in a target address cache. By the time a backward branch threshold of an address has been reached, the interpreter switches to a mode where instructions are recorded at the same time as they are executed. This creates a single-entry, multi-exit linear sequence of instructions called a "trace". Results show that this approach is able to optimize opportunities that manifest themselves only at runtime.

Both [AK87] and [ZC90] have laid the foundation for loop transformation into vectorized or parallel form. To transform loops into a semantically equivalent vector form is the data dependence. Strongly Connected Components (SCCs) are built from the data dependency graph and aid to distribute the loop partly or totally into vector form.

Vectorization as an optimization technique in JIT compilers is seldom. One research project extended the Jikes RVM [ESEMEN09] to automatically vectorize loops. The technique that used an extended tree pattern matcher was not improved by follow up projects. Others [RDN$^+$11] try to find data parallelism on byte code level by annotating information that can later used by the JIT VM. [LCF$^+$07] is another example to annotate the byte code generated to enable the VM to vectorize loops.

3 PyPy's JIT

When we speak of PyPy's JIT compiler, we really mean the effort put into the RPython tool chain. The fundamental idea of RPython is to provide both a JIT compiler and garbage collector to any dynamic language written in RPython. Thus it is not only a subset of the Python programming language but also a tool chain to ease the construction of byte code interpreters. The translation itself is a vast topic. It is not the main topic of this document, thus information can be found in e.g. [BCFR09] or [BR07].

The tracing JIT compiler is generated for the main interpreter loop dispatching the byte codes. The only addition required to the interpreter annotates the dispatch header and the backwards jump. An automatic process creates an abstract representation that can be traced and JIT compiled. Figure 1 shows a sample trace tree.

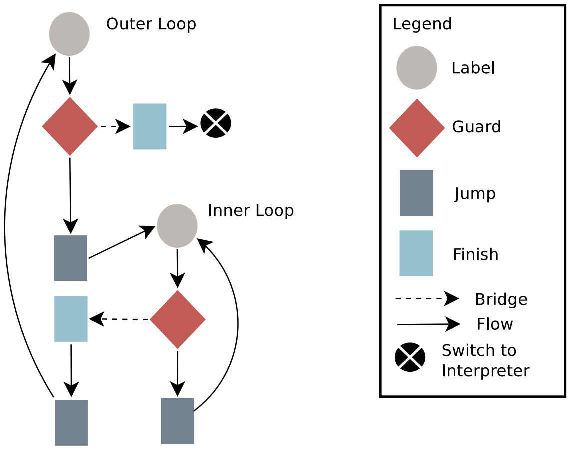


Fig. 1. A trace tree constructed by e.g. PyPy's tracing interpreter. It shows a doubly nested loop.

It represents a nested loop, switching to the inner loop in the middle of the outer loop. Guarding instructions ensure the correctness of the execution. Whenever a guard fails frequently, a "bridge" is created and attached to the trace. To exit a trace loop the bridge ends in a "Finish" operation and continues to execute an outer loop or switches back to the interpreter.

The instructions that form the outer loop body are split by the inner loop. Operations prior the inner loop are executed from the outer loop header until entering the inner loop by a "Jump"[1] operation. The guard exit leading into a "Finish" operation executes all operations that succeed the inner loop until the outer loop is closed again.

4 Contributions to PyPy's TJIT

The following contributions have been made to the tracing just-in-time compiler backend and it is now able to:

- Create a dependency graph for trace instructions.
- Unroll a trace loop for a factor greater than two.
- Find, extend and combine groups of parallel instructions.

[1] In RPython, this jump is named "call assembler" and is a different operation than the jump to a loop header.

– Schedule a dependency graph and emit vector statements.
– Strengthen guards that protect comparison.
– Create several different version of the trace loop and stitch it to guard instructions.
– Support accumulation patterns (e.g. sum).
– Emit SSE4.1 machine code for vector instructions.

Henceforward the term "VecOpt" will refer to both the branch that includes the changes[2] and the implemented algorithm. Section 8 refers to VecOpt as a compiled PyPy interpreter using the contributions of this document.

5 Motivation

PyPy is eager to provide parts of the NumPy library within the standard library of their virtual machine. At the time of writing one of the biggest challenge is to compete with the speed of native code produced by an ahead of time compiler for NumPy kernels. It was decided to reimplement part of the library due to major limitations.

– Many array operations invoke foreign functions. The penalty can be significant for PyPy.
– They are written and must be maintained in a low level language (e.g. Fortran,C).
– By reason of the moving garbage collector, there is no API to let foreign code access PyPy's internal objects. This is one of the biggest limitation that separates CPython and PyPy.

The native NumPy routines used by CPython are written in C and use the CPython API to manipulate Python objects. It uses a preprocessing utility[3] to generate all numerical kernels and use plain memory/pointer arithmetic to access elements. The loop kernels are unrolled manually to ensure that SIMD operations are emitted by the ahead of time compiler.

The numerical kernels of PyPy are written in RPython using an iterator API to access memory elements. The numerical kernels are parameterized with the kernel function, operator types and result type. They take full advantage of the tracing JIT compiler.

6 Design

Program transformations for vector machines try to maximize the size of vectors to be processed in parallel. The resulting parallel execution improves the bigger the input vectors. Statements and the loop nest provide the basic information

[2] Located at https://bitbucket.org/pypy/pypy. Aug. 2015.

[3] It does not use the preprocessor to duplicated routines for different element types. The preprocessor is annotated in comments.

to build a cyclic dependency graph. Strongly connected components (SCC) are identified and the graph's topological order is used to emit vector statements that are not contained in SCCs. SIMD instructions have a bounded vector size thus the usual abstractions force the code generation to split up the vectors into short vectors again.

In a tracing context the nesting of a loop is opaque and the inner most loop is always traced first. This limitation is a design decision that helps to cope with one problem object oriented languages impose on the runtime: abstraction through layering. A function call often flows through several object layers to accomplish small tasks. The well known optimization to improve performance in these cases is called "Inlining". A tracing compiler can efficiently inline and optimize the execution. At the same time the assembled machine code size of a trace is only a fraction compared to a method base compiler.

Practically speaking, the abstractions for nested loops and acyclic dependency construction are well suited for vector machines. Whenever time is of essence and the vector size is bounded a different approach might yield similar results. The algorithm proposed by Larsen [LA00] is able to vectorize basic blocks.

Parallel instructions are gathered by unrolling the loop. Dependency construction is simplified because cyclic dependencies are ignored. Only loop independent dependencies are tracked using the definition use chains of the basic block. This can be done in a linear pass over the trace loop using a associative data structure to remember definitions. Opposed to this, approaches like the Power test [WT92], the Omega test [Pug91] or the well known GCD [Ban97] test need linear/affine equations and solvers to determine the dependency.

The rest of the algorithm boils down to a scheduling problem. The dependency graph is used to group independent and isomorphic instructions. This information is then considered while rescheduling the trace and emits vector instructions.

7 Superword parallelism on trace sequences

The optimization routine is outlined in Algorithm 1. Although the the implementation in the RPython optimization backend is quite similar to [LA00] and [PKH07] there are some key differences.

Algorithm 1 shows the preparation routine for a trace loop and the algorithm to vectorize trace loops. The function BasicInfo returns the smallest type in bytes (for load/store operations), a list of operations that reference memory (read/write) and all modifications on index variables. The three different information types can be acquired in a single forward pass. The unrolling factor is heuristically determined by the smallest type and the size of the vector register. The smallest type has been chosen, to offer more opportunity to pack instructions. By choosing the biggest type, occasionally packed instructions do not span over the whole vector register.

Tracing checks the loop index at the end of the trace, before it jumps back to the header. This check at the end adds an dependency to the next load instruction and the previous store instruction of the unrolled trace loop. It is impossible to execute the instructions in parallel. Relax in Algorithm 1 finds the index guards and moves them to the beginning of the loop. This operation is then marked as an "early exit" which enables the dependency builder to reduce the dependencies.

Algorithm 1 Vectorization optimization routine

```
T ... Trace loop
vs ... Size of the hardware vector register
$M_r$ ... Set of instructions that read/write memory references
$I_v$ ... Set of affine combinations for indices
function Prepare(T,vs)
    T ← Relax(T)
    b, $M_r$, $I_v$ ← BasicInfo(T)
    factor ← $\frac{vs}{b}$
    $T_u$ ← unroll(T,factor)
    return ($T_u$,$M_r$,$I_v$)
function Vectorize(T, $M_r$, $I_v$)
    G ← BuildDependecyGraph(T, $I_v$)
    $P$ ← InitPairs(G, $M_r$, $I_v$)
    $P$ ← Extend($P$, G)
    $P$ ← Combine($P$)
    $T_{vec}, savings$ ← Schedule(G, P)
    if savings $\leq -1$ then
        return T
    return $T_{vec}$
```

The output of Prepare is the input for Vectorize. I_v is used to determine if memory loads/stores alias or if they are adjacent in memory e.i. Adjacent. Without inferring this information, the resulting dependency graph cannot assume that two memory stores don't depend on each other. This introduces edges which are not necessary in most cases, but prohibit vectorization.

InitPairs, Extend and Schedule are shown in Algorithm 2,3,4 respectively.

7.1 Initialize and Extend

InitPairs create pairs of adjacent memory operations that are both isomorphic and independent. Isomorphic is defined as "semantically equivalent intermediate instruction". Relying on these properties, a parallel execution is semantically valid.

Extend enumerates all known pairs and tries to follow the definition and use chains. The Cartesian product of the two calls to Def/Use represent the

Algorithm 2

```
function InitPairs(G, $M_r$, $I_v$)
    $P \leftarrow \emptyset$
    for $m_1, m_2 \in M_r \times M_r$ do
        if Adjacent($m_1, m_2$) $\wedge$ Isomorphic($m_1, m_2$) $\wedge$
            Independent(G,$m_1$,$m_2$) then
            $P \leftarrow P \cup$ Pair($m_1$,$m_2$)
```

instructions combinations possible for two pairs. These candidates are subject of extending the list of pairs. The clou of this algorithm is to find the pairs that directly use or input pairs to the same argument slots. If the operation has a vectorized equivalent, a hardware SIMD instruction might be able to execute the operation faster. The routine continues as long as new candidate pairs are found.

Algorithm 3

```
function Extend(P, G)
    $C \leftarrow \emptyset$
    while $C \neq |P|$ do
        $C \leftarrow |P|$
        for Pair($i_1, i_2$)$\in P$ do
            for $i_3, i_4 \in$ Use(G,$i_1$) $\times$ Use(G,$i_2$) do
                if Isomorphic($i_3, i_4$) $\wedge$ Independent(G,$i_3$,$i_4$) then
                    $P \leftarrow P \cup$ Pair($i_3$,$i_4$)
            for $i_3, i_4 \in$ Def(G,$i_1$) $\times$ Def(G,$i_2$) do
                if Isomorphic($i_3, i_4$) $\wedge$ Independent(G,$i_3$,$i_4$) then
                    $P \leftarrow P \cup$ Pair($i_3$,$i_4$)
```

7.2 Combine and Schedule

Up to this point only pairs of operations have been recorded. By design pairs can overlap with other pairs. Given the two pairs (l_1,l_2) and (l_2,l_3) they can be merged into a pack of three elements (l_1,l_2,l_3). This task is accomplished by Combine. It has been omitted from the listing, since it's implementation is straight forward. It simply compares pack by pack and merges them if the right most operation matches the left most. It already takes into account the vector size provided by the target ISA and stops to pack further operations if the limit of the vector size is reached.

To accomplish tight packing and the minimum number of resulting packs the input pairs are sorted. Each pair's first operation is sorted ascending. The current pack is expanded as long as there are more matching packs and the capacity has been reached.

In the last step the trace is rescheduled using the information gathered earlier. The scheduling algorithm is interwoven with logic to estimate the savings of the loop. The estimated savings for packing an instruction is modeled using the CPU architecture in mind. The basic saving can be calculated using the following formula: $s = -cost + count(pack) * \text{benefit}$. E.g. The SSE4.1 instruction ADDPD is modeled as $s = -1 + 2 * 1 = 1$.

UnpackCost models the costs needed to unpack variables that are contained in any vector registers. Depending on the position the function estimates costs modeled after the CPU architecture. E.g. Unpacking the higher element of a double precision floating point has a higher cost than unpacking the lower element[4].

Scheduling picks a candidate operation that is scheduleable. An operation in the dependency graph is schedulable if there are no edges that point to the operation. This is trivially true for the label operation, which starts the scheduling.

If the candidate operation to be scheduled has an associated pack, all operations are transformed to a single vector operation by VectorOperations. For this to succeed all operations of the pack must be schedulable, otherwise the current candidate is postponed. Then all edges to descending operations (e.i. the ones that depend on the current operation) are removed in Scheduled. A call to Next gathers all operations that are now schedulable after edges have been removed.

7.3 Enhancements

Scalar constants and variables are expanded. If the scalar value is produced in the loop, the expansion creates the vector register before it is used. In any other case a dedicated vector register is reserved before the trace loop is entered. The constant or variable content is scattered to each slot of the vector register. The operation is later able to use the expanded register instead of executing the loop iterations one by one.

Accumulation of values (e.g. sum,product) can also be transformed into vector instructions. The summation of a vector contains dependent addition instructions for a value that is carried across the trace loop. This pattern is recognized and a special pair is added in Expand. Similar to variable expansion the accumulator is expanded before the loop is entered. The summation is done using a normal vector addition. Parts of the sum are accumulated at the slots of the vector register. After exiting the loop through any guard the vector register is added horizontally to a single value. This transformation is only valid for commutative operations such as addition or multiplication as well as logical reductions as and ($\wedge$), or ($\vee$) or xor ($\oplus$).

[4] The assembler backend needs at least 2 assembler instructions for the high element, instead of a maximum of one for the lower element.

Algorithm 4

```
function SCHEDULE(P, G)
    $S \leftarrow 0$
    $T \leftarrow \emptyset$
    $N \leftarrow$ NEXT(G,$\emptyset$)
    while $N \neq \emptyset$ do
        $O \leftarrow$ HEAD(N)
        pack $\leftarrow$ PACK(P,O)
        if $\neg$ pack then
            $T \leftarrow T \cup \{O\}$
            $S \leftarrow S -$ UNPACKCOST(O)
            SCHEDULED(G,O)
        else
            if PACKSCHEDULEABLE(pack) then
                $S \leftarrow S -$ PACKCOST(pack)
                $T \leftarrow T \cup$ VECTOROPERATIONS(pack)
                $S \leftarrow S +$ ESTIMATESAVINGS(pack)
                SCHEDULED(G,pack)
            else
                $N \leftarrow N \ \cup \{O\}$
        $N \leftarrow$ NEXT(G,N)
    return T,S
```

8 Evaluation

The evaluation is split into two different parts. The first measures the time spent in the trace loops that have been vectorized and is compared to the scalar trace loops. The second are programs that do not stress the vectorization algorithm, but try to evaluate the gain the optimization is able to achieve.

Although the implementation is already nearly finished, these benchmarks are a preview for the final implementation.

8.1 Trace loop benchmarks

The following programs have been evaluated using the following configuration: Intel i7-4550U CPU @ 1.50GHz with 4 cores, Linux Kernel 4.0.6.

The source code can be found in the branch "vecopt" and is based on the PyPy release version 2.6.0. The garbage collector "incminimark" was prevented to be run in the trace loop benchmarks by setting the minimum memory threshold to 4GB of allocated memory. Below the modified threshold, the garbage collector does not start a collection run.

For the following measurements, the tracer and JIT compiler has been instrumented[5] to measure the time elapsed in traces. The function to time the execution was *clock_gettime*. It records the CPU time spent in the process.

[5] The revision *a026d96015e4* was used for this benchmark run. It imposes a significant performance penalty when exiting or entering traces.

Table 8.1 shows the micro seconds that have been spent in the optimization pass. It excludes all other optimizations.

Table 1. Optimization time measured. Instruction count is the number before the transformation and unrolling has been applied.

Count	Instruction count	Unroll factor	Microseconds	Variance
6	12-16	2	101.47	9.90
5	17-19	4	158.46	4.57
2	17	8	224.03	2.20
2	17	16	396.60	1.24

Figure 2 shows several different vector calculations. The horizontal line shows the baseline of the normal trace. Every program run iterates the operation for 1000 execution. The vector operands are sized four times the tracing threshold. The following listing shows a sample program that is used in Figure 2.

```
def bench(vector_a, vector_b):
  for i in range(1000):
    numpy.multiply(vector_a, vector_b, out=vector_a)
```

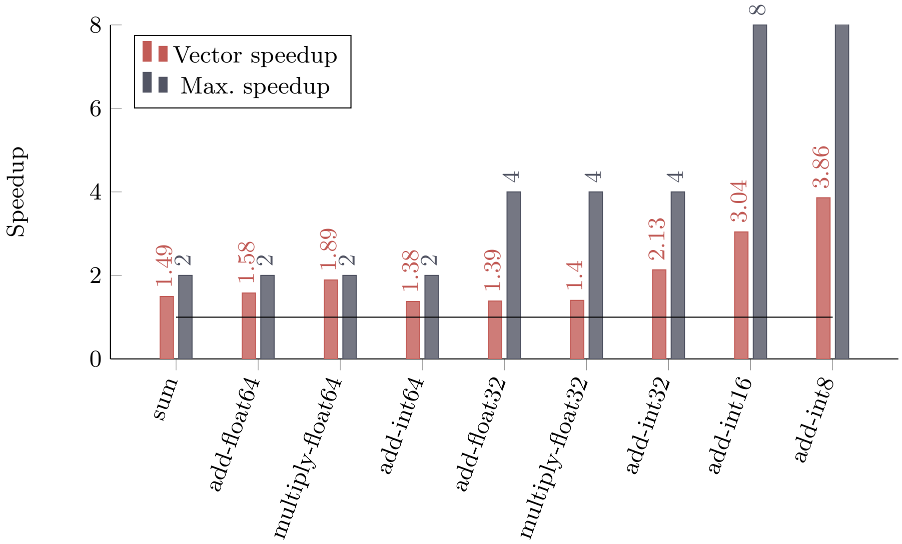


Fig. 2. Speedup of the vectorized trace loops. Horizontal line is the baseline for the calculated speedup values ($speedup = \frac{scalar}{vector}$).

Single floating point operations don't show a significant speedup to their scalar trace loops. The reason for this behavior is that floating point operations are always done on the biggest floating point type available. The language semantics of Python make the size of floating point numbers platform specific, thus the tracer does not emit floating point operations for single floats, but casts them to double floats.

The theoretical maximum speedup can be observed for loops with double float multiply operations. Other loops show about half of the expected speedup. Considering that it is currently not possible to use aligned vector statements the results are quite satisfying.

Integer addition for 16/8 bits don't show very good results due to the small vector size. It has been observed that on bigger vector sizes these data types perform better. In any case these instructions are not expected to be used very frequently in NumPy programs.

8.2 NumPy benchmark suite

The following evaluates VecOpt on small to medium sized numerical kernels. The latter configuration is a mobile CPU chip. For these benchmarks a hardware configuration was used that offers more performance. Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz, 4 cores, Linux 4.1.5. Python 2.7.10 and NumPy version v1.9.2rc1 has been used as a base line implementation. VecOpt uses revision **3742fae37** and the forked NumPy branch for PyPy (**504ee4757**). PyPy uses the 2.6.0 release binary (**295ee98b69**).

Table 3 shows a NumPy benchmark suite[6]. The source code was forked and modified. The modifications executes the kernel several times to warm up the JIT compiler. Each benchmark is repeated five times and the mean value is displayed in the table. For PyPy the benchmark kernel is executed twenty times in the warm up phase. Table 2 shows the loop count of the kernels.

Name	Loop	Warm up
diffusion	20	5
allpairs-distances	30	20
vibr-energy	100	20
l2norm	100	20
rosen	30	10

Table 2. The loop count and warm up iteration count for the benchmark programs in 3. All kernels that are not listed loop 50 times and warm up 20 iterations.

Table 3 shows that for some benchmarks only minor improvements can be achieved. The current weakness both PyPy and VecOpt suffers from is related

[6] https://github.com/planrich/numpy-benchmarks. Aug. 2015

Name	CPython (C_1)	PyPy (C_2)	VecOpt (C_3)	$Speedup\frac{C_1}{C_3}$	$Speedup\frac{C_2}{C_3}$
allpairs-distances	0.9868	2.57	2.534	0.39	1.0
allpairs-distances-loops	1.826	4.287	4.177	0.44	1.0
arc-distance	0.07898	0.1813	0.1608	0.49	**1.1**
diffusion	0.5603	5.665	3.889	0.14	**1.5**
evolve	0.1967	1.815	1.728	0.11	**1.1**
fft	0.9507	0.2981	0.2955	3.2	1.0
harris	0.3485	3.119	1.504	0.23	**2.1**
l2norm	0.564	1.73	1.634	0.35	**1.1**
lstsqr	0.3844	1.506	1.39	0.28	**1.1**
multiple-sum	0.1432	0.6341	0.5768	0.25	**1.1**
rosen	0.5795	3.498	3.438	0.17	1.0
specialconvolve	0.4713	3.876	2.649	0.18	**1.5**
vibr-energy	0.2784	0.7552	0.699	0.4	**1.1**
wave	2.191	1.114	1.166	1.9	0.96
wdist	2.927	1.202	1.179	2.5	1.0

Table 3. Benchmark suite. C_1, C_2 and C_3 show the CPU clock time spent. C_4 and C_5 show the speedup. C_5 additionally marks the improvements introduced by VecOpt.

to the allocation of memory in the benchmark kernel. CPython's GC uses reference counting which immediately frees NumPy arrays. PyPy's GC might keep memory for many more cycles. In Section 8.3 we will see custom written kernels, that do not allocate memory within the kernel loop.

Table 3 indicates that CPython most of the time is a better choice than PyPy. The only reason why CPython has such good results is because a significant fraction of time is spent in native code, removing all interpretative overhead. Furthermore note that the NumPyPy library has not completely implemented all features offered by NumPy.

8.3 Pure Python loops and other kernels

To show that there are really more significant improvements than presented in the previous section, a list of benchmarks has been compiled[7]:

- **som** - Self Organizing Maps[8].
- **dot** - Matrix vector dot product.
- **any** - Micro benchmark stressing the any NumPy operation.
- **fir*** - Finite impulse response.

[7] https://github.com/planrich/pypy-simd-benchmark Aug. 2015

[8] This implementation is not complete. It only simulates the "find nearest neighbor" and "update weight vector" step of the algorithm. Is a numeric application that makes heavy use of vector subtractions, multiplications, distance and summation. Similar to principal component analysis this procedure can be employed as a pre step for machine learning.

- **add*** - Addition of a Python array.
- **sum*** - Summation of a Python array.
- **rgbtoyuv*** - RGB to Y'UV converions using Python arrays.

All benchmarks that end with an asterisk symbol (*) are pure Python implementations. Indeed the optimizer makes no distinction between NumPy and Python traces, but is currently by default deactivated for the latter.

Name	Vector size	Repeat count
som	256	4000
dot	1000	1000
any	1024	1000
add*	2500	10000
sum*	2500	10000
fir*	200	3000
rgbtoyuv*	$1024 * 768$	500

Table 4. The vector size and the repetition count of the kernel benchmark programs in Figure 3. All programs are run ten times and the mean value is used to calculate the speedup value.

9 Future work

PyPy is constantly changed and improved. Only recently work has been started to cut down the optimization time and improve the warm up speed of the virtual machine. Interestingly these changes already track the dependencies of the IR operations and with little effort, the dependency construction step can be merged with the new model. There are plans to integrate these changes and enable them by default with the new optimization setup.

One deficiency has already been mentioned in the evaluation section. Allocating memory frequently is not handled very well by PyPy's GC. Moreover it does not know that the allocated NumPyPy array is a large chunk of memory, but only sees the object encapsulating the pointer to the actual storage. This problem could be mitigated by changing parts of the NumPyPy library.

The unrolling heuristic performs very well for most numerical loops. But it is ignoring the fact that there could be already several memory load/store instructions that could be grouped to one vector operation.

Furthermore there are cases where the input memory locations are interleaved. If follow up instruction wanted to use the vector register, they would need to shuffle the slots to continue. This is not done in the current implementation.

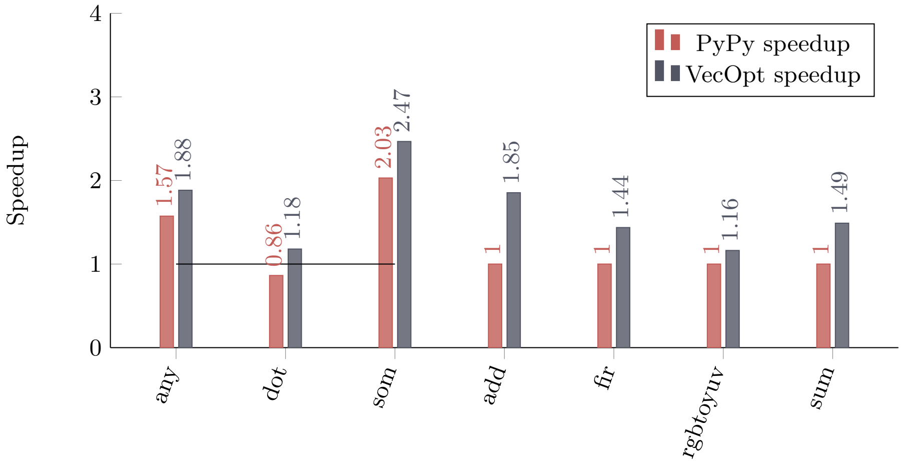


Fig. 3. Yet another benchmark plotting the speedup. The first four runs use CPython as base line to measure the speedup (Indicated by the horizontal line). For all others CPython had to be excluded from the benchmark run. All of them are written in pure Python. CPython is not able to execute any computation in native code and thus takes far to long to complete the benchmark run. The speedup of VecOpt in these cases uses PyPy as baseline implementation. Due to some limitations of the current prototype, RGB to YUV operations on floating points rather than 8/16 bit bytes.

10 Conclusion

It has been shown that a tracing JIT compiler can indeed use SIMD instructions to speed up numerical loops. This is not only true for the NumPyPy standard library, but also for any other traces that adheres the pattern the transformer understands. It additionally shows that the optimization time is reasonably fast and the implementation complexity is rather low. The contributions do not only enhance PyPy, but for any other virtual machine written in RPython. This opens up new possibilities to write a virtual machine that executes numerical computations fast using all the comfort a dynamic language provides.

References

[AK87] Randy Allen and Ken Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9:491–542, 1987.

[Ban97] Utpal Banerjee. *Dependence analysis*, volume 3. Springer Science & Business Media, 1997.

[BCFR09] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization* *of Object-Oriented Languages and Programming Systems*, pages 18–25. ACM, 2009.

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *ACM SIGPLAN Notices*, volume 35, pages 1–12. ACM, 2000.

[BR07] Carl Friedrich Bolz and Armin Rigo. How to not write virtual machines for dynamic languages. In *3rd Workshop on Dynamic Languages and Applications*, 2007.

[ESEMEN09] Sara El-Shobaky, Ahmed El-Mahdy, and Ahmed El-Nahas. Automatic vectorization using dynamic compilation and tree pattern matching technique in Jikes RVM. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 63–69. ACM, 2009.

[LA00] Samuel Larsen and Saman Amarasinghe. *Exploiting superword level parallelism with multimedia instruction sets*, volume 35. ACM, 2000.

[LCF$^+$07] Piotr Lesnicki, Albert Cohen, Grigori Fursin, Marco Cornero, Andreas Ornstein, and Erven Rohou. Split compilation: an application to just-in-time vectorization. In *Workshop on GCC for Research in Embedded and Parallel Systems (GREPS'07)*, Brasov, Romania, 2007.

[PKH07] Ivan Pryanishnikov, Andreas Krall, and Nigel Horspool. Compiler optimizations for processors with SIMD instructions. *Software: Practice and Experience*, 37(1):93–113, 2007.

[Pug91] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13. ACM, 1991.

[RDN$^+$11] Erven Rohou, Sergei Dyshel, Dorit Nuzman, Ira Rosen, Kevin Williams, Albert Cohen, and Ayal Zaks. Speculatively vectorized bytecode. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, pages 35–44. ACM, 2011.

[WT92] Michael Wolfe and Chau-Wen Tseng. The power test for data dependence. *Parallel and Distributed Systems, IEEE Transactions on*, 3(5):591–601, 1992.

[ZC90] Hans Zima and Barbara Chapman. *Supercompilers for parallel and vector computers*. ACM, 1990.

Java Type System – Proposals for Java 10 or 11

Martin Plümicke

Baden-Wuerttemberg Cooperative State University Stuttgart/Horb
Department of Computer Science
Florianstraße 15, D–72160 Horb
pl@dhbw.de

Abstract. In this paper we will present ideas for the extension of the Java type system. On the one hand Java could get real function types. There are some disadvantages of the Java 8 approach to use target types as types for lambda expressions. In our approach the idea of target typing is preserved but extended by real function types. From this extension follows an extension of our type inference algorithm.
On the other hand we extend the Java type system by intersection types of function types. The principal types of functions in Java are in general intersection types.

Introduction

The development of Java in the last decade has introduced many features from functional programming languages. While in Java 5.0 [GJSB05] generics are introduced in Java 8 [GJS$^+$14] lambda expression are added. In [Plü07,Plü15] we proposed Java type inference systems that allows to give Java programs without type annotations. Type inference systems are also well-known from functional programming languages.
All these three approaches have some difficulties but were good enough. We address these difficulties in this paper. For this we extend the Java type system again. We call the language Java Type Extended (Java-TX), that is a conservative extension of Java 8.
In Java 8 lambda expressions themselves have no explicit types. They get as target types so-called functional interfaces (interfaces with one method) from the context. This approach has the advantage that many implementations of existing call-back interfaces are improved. But it has also some disadvantages i.e. the subtyping property. Therefore in Java-TX we add a concept of real function types as explicit types of lambda expressions. For this we define a set of special interfaces Fun$N$*, that represent real function types. We address this extension in Section 1.
In Section 2 we explain the role of the Fun$N$*–types in our type inference system. The inferred types of Java functions are in general intersections of function types. As Java allows no intersection types, the intersections had to be resolved by the programmer. Since now, we do this by an eclipse plugin [Sta15]. In Java-TX

we introduce intersection types of function types. In Section 3 this extension is addressed.
Finally, we close with a conclusion and give an outlook.

1 Real function types

In the past we considered two different type inference algorithms for lambda expressions. While in [Plü11] real function types are considered, in [Plü15] the Java 8-like functional interface are used. In Java-TX we merge these both approaches, as both have some advantages.

1.1 The special interface Fun$N$*

A lambda expression in Java 8 has no explicit type. The type is determined by the compiler from the context in which the expression appears. This means that one lambda expression can have different types in different contexts.

```
Callable<String> c = () -> "done";
PrivilegedAction<String> a = () -> "done";
```

In the first context for the lambda expression the type `Callable<String>` is determined, while in the second context `PrivilegedAction<String>` is determined.
In [Plü14] we summarized all functional interfaces to equivalence classes, which single abstract method's have the same typings. As a representation of the respective classes we introduce for simulating function types a predefined collection of interfaces for all $N \in \mathbb{N}$:

```
interface FunN<R,T1 , ... ,  TN> {
    R apply(T1 arg1 , ... ,  TN argN);
}
```

The following example shows the inconvenience of this approach.

Example 1. Let be the following function g defined:

```
g = x -> y -> f -> f.apply(x,y);
```

The curried function g takes three arguments, where the third argument is a function, that is applied to the first and the second argument. In a functional programming language a principal type of g would be

```
A -> (B -> (((A, B) -> C) -> C)).
```

But with the Fun$N$-construction the equivalent type would be

```
Fun1<? extends Fun1<? extends Fun1<? extends C,
                                   ? super Fun2<? extends C,? super A,? super B>>,
                    ? super B>,
     ? super A>
```

Nearly no programmer would give g such type, although it is the principal type.

In Java-TX we extend these interfaces to special interfaces Fun$N$*, where the subtyping property is changed in comparison to Java. The special interfaces Fun$N$* correspond to functions types in Scala [Ode14].

The language Java-TX contains interfaces for all $N \in \mathbb{N}$

```
interface FunN*<+R,-T1, ... , -TN>¹ {
    R apply(T1 arg1, ... , TN argN);
}
```

where Fun$N$*<$\mathtt{T}_0$,$\mathtt{T}'_1, \ldots, \mathtt{T}'_N$> $\leq^*$ Fun$N$*<$\mathtt{T}'_0$,$\mathtt{T}_1, \ldots, \mathtt{T}_N$> iff $\mathtt{T}_i \leq^* \mathtt{T}'_i$ with $\leq^*$ as subtyping relation. For Fun$N*$ no wildcards are allowed.
Let us consider the following example

```
Object m(Integer x, Fun1*<Object, Integer> f) {
  return f.apply(x);
}
```

It is obvious, that the following application is correct:

```
Fun1*<Object,Integer> f_IntObj = ...
Object x2 = m(2, f_IntObj);
```

But for Integer $\leq^*$ Number $\leq^*$ Object also

```
Fun1*<Integer,Integer> f_IntInt = ...
Object x1 = m(2, f_IntInt);
```

is correct, as Fun1$*$<Integer, Integer> is a subtype of Fun1$*$<Object, Integer> and

```
Fun1*<Number, Number> f_NumNum = ...
Object x3 = m(2, f_NumNum);
```

is correct, as Fun1$*$<Number, Number> is also a subtype of Fun1$*$<Object, Integer>.

Example 2. Considering again Example 1 the program

```
        g = x -> y -> f -> f.apply(x,y);
```

has in Java-TX the type Fun1*<Fun1*<Fun1*<C,Fun2*<C,A,B»,B>,A>

1.2 Fun$N*$ as types of methods

We can also give Fun$N*$–types to methods. This means with the class CL

```
class CL {

   T0 meth (T1 x1, ... , TN xN) { ...  }
}
```

the method reference CL::meth has the type Fun$N$$*$<$\mathtt{T}_o$,$\mathtt{T}_1, \ldots, \mathtt{T}_N$>.
The advantage of this definition is that method references can be used as lambda expression. Also subtyping and direct applications work in the same manner.

[1] The arguments are covariant resp. contravariant, written as in Scala [Ode14]

1.3 Integration of real function types into Java-8

We preserve in our approach the great benefits of the target typing in Java 8 by integration both concepts. The target typing is extended in the following way:

- A lambda expression itself has an explicit Fun$N$*–type.
- A lambda expression fits any target type, which must be a functional interface, if its method's type in Fun$N$*–representation is a supertype of the explicit type.

Example 3. Let us consider again:

```
Callable<String> c = () -> "done";
PrivilegedAction<String> a = () -> "done";
```

The explicit type of the lambda expressions `() -> "done"` is `Fun0∗<String>`. The types of the methods `call` of `Callable<String>` and `run` of `Privileged-Action<String>` have also the type `Fun0∗<String>`. This means that the target types are compatible.

2 Type inference

Another feature well-known from functional programming languages is type inference. In object-oriented languages, type inference is only in the restricted form of local type inference [PT98] implemented, while in Java 8 some elements are introduced. It is possible to leave out the argument types of lambda expression (instead `(ty a) -> expr` it is possible to write `(a) -> expr`). Furthermore the so-called diamond operator is introduced. This means that it is possible to write `new Class<>` and the parameters of `Class` are inferred.

But complete type inference, especially type inference of recursive declared functions is not implemented.

The main reason for this lack is that the results in the defined Java type system are generally not unique.

We address this problem in different approaches. In [Plü07] we gave a type inference algorithm for Java with generics including wildcards. In [Plü11] we presented a type inference algorithm for Java with real function types. In [Plü15] finally we presented a type inference algorithm for Java with lambda expressions and functional interfaces.

In this section we present the type inference algorithm for Java-TX. For this we have to combine the approaches of type inference for real function types [Plü11] and type inference for functional interfaces [Plü15]. Java-TX uses the special interfaces Fun$N$* for function types, that are nominal types. Therefore we use the base of [Plü15]. The differences in the results are solved by adapting the underlying type unification [Plü09].

2.1 The algorithm

The type inference algorithm (Figure 1) takes a set of type assumptions `TypeAssumptions` and a untyped class `Class` and gives a pair of a set of remaining constraints `Constraints` and a typed class `TClass`.

$$
\begin{array}{l}
\mathbf{TI}\colon \texttt{TypeAssumptions} \times \texttt{Class} \rightarrow \{\, (\texttt{Constraints}, \texttt{TClass})\,\} \\
\quad \mathbf{TI}(\, \mathit{Ass}, \mathsf{Class}(\, \tau, \mathsf{extends}(\, \tau' \,), \mathit{fdecls}\,)\,) = \\
\qquad \mathbf{let}\ \underline{(\mathsf{Class}(\, \tau, \mathsf{extends}(\, \tau' \,), \mathit{fdecls}_t \,), \mathit{ConS})} = \\
\qquad\qquad\quad \mathbf{TYPE}(\, \mathit{Ass}, \mathsf{Class}(\, \tau, \mathsf{extends}(\, \tau' \,), \mathit{fdecls}\,)\,) \\
\qquad\qquad \underline{\{\, (cs_1, \sigma_1), \ldots , (cs_n, \sigma_n)\,\}} = \mathbf{SOLVE}(\, \mathit{ConS}\,) \\
\qquad \mathbf{in}\ \{\, (cs_i, \sigma_i(\, \mathsf{Class}(\, \tau, \mathsf{extends}(\, \tau' \,), \mathit{fdecls}_t \,)\,))|\ 1 \leqslant i \leqslant n\,\}
\end{array}
$$

Fig. 1. The type inference algorithm

TI consists of two main functions **TYPE** and **SOLVE**, where **TYPE** inserts type annotations, widely type variables as placeholders, in the Java class and determines a set of type constraints and **SOLVE** solves the constraints by our type unification algorithm [Plü09]. The result of **SOLVE** is a set of pairs $\{\, (cs_1, \sigma_1), \ldots , (cs_n, \sigma_n)\,\}$, where the cs_i consists of remaining constraints $(a \lessdot a')$ of types variables and σ_i consists of solutions $(a \doteq \theta)$, where $(a \lessdot a')$ means a has to be a subtype of a' and $(a \doteq b)$ means a and b are equal.

Let us consider the class `Matrix` in Figure 2. A class `Matrix` is declared as an extension of `Vector<Vector<Integer>>`. `op` is a function defined by a lambda expression in curried representation with two arguments. First it takes a matrix and second it takes a function, that has as arguments two matrices and returns another matrix. The result of `op` is the application of the function (second argument) to its object (`this`) and its first argument. The method `mul` is the ordinary matrix multiplication in lambda representation. Finally, in `main` the function `op` is applied. The `op`-function of matrix `m1` is applied to the matrix `m2` and the function `mul` of `m1`. In the figure the class `Matrix` is shown in Java 8 and in Java-TX. The Java-TX program shows the possibilities to declare programs without type annotations. A little curious is the declaration of local variables `ret; v1; v2; m1;` and `m2;`. This is necessary as for the reason of unambiguousness Java-TX retains the Java property that all variables must be declared before used.

2.2 Type unification

In the function **SOLVE** the type unification is called to solve the type constraints. In [Plü09] we described the type unification for the Java type system. The introduction of the `Fun`N^* types induces an extension of this unification. The three most important added unifications rule are given in Figure 3. In the rules $a \lessdot b$ means a must be a subtype of b and $a \doteq b$ means a and b must be equal.

```
//Java 8 with type annotations
class Matrix extends Vector<Vector<Integer>> {

  Fun1<Fun1<Matrix, Fun2<Matrix, Matrix,Matrix>>, Matrix>
  op = (Matrix m) -> (Fun2<Matrix, Matrix,Matrix> f) -> f.apply(this, m);

  Fun2<Matrix, Matrix,Matrix> mul = (Matrix m1, Matrix m2) -> {
      Matrix ret = new Matrix ();
      for(int i = 0; i < size(); i++) {
          Vector<Integer> v1 = m1.elementAt(i);
          Vector<Integer> v2 = new Vector<Integer> ();
          for (int j = 0; j < size(); j++) {
              int erg = 0;
              for (int k = 0; k < v1.size(); k++) {
                  erg = erg + v1.elementAt(k).intValue()
                    * (m2.elementAt(k)).elementAt(j).intValue(); }
                  v2.addElement(erg); }
              ret.addElement(v2); }
          return ret; };

    public static void main(String[] args) {
        Matrix m1 = new Matrix(...);
        Matrix m2 = new Matrix(...);
        (m1.op.apply(m2)).apply(m1.mul);}
}
```

```
//Java-TX without type annotations
class Matrix extends Vector<Vector<Integer>> {

    op = (m) -> (f) -> f.apply(this, m);

    mul = (m1, m2) -> {
        ret; ret = new Matrix ();
        for(int i = 0; i < size(); i++) {
            v1; v1 = m1.elementAt(i);
            v2; v2 = new Vector<Integer> ();
            for (int j = 0; j < size(); j++) {
                int erg = 0;
                for (int k = 0; k < v1.size(); k++) {
                    erg = erg + v1.elementAt(k).intValue()
                      * (m2.elementAt(k)).elementAt(j).intValue(); }
                 v2.addElement(erg); }
            ret.addElement(v2); }
        return ret; };

    public static void main(String[] args) {
        m1; m1 = new Matrix(...);
        m2; m2 = new Matrix(...);
        (m1.op.apply(m2)).apply(m1.mul);}
}
```

Fig. 2. Matrix in Java 8 respectively in Java-TX without th and type annotations

$$(\text{reduceFun}N{*})\ \ \frac{Eq \cup \{\, \texttt{Fun}N\texttt{*<}\theta, \theta'_1, \ldots, \theta'_N\texttt{>} \lessdot \texttt{Fun}N\texttt{*<}\theta', \theta_1, \ldots, \theta_N\texttt{>} \,\}}{Eq \cup \{\, \theta \lessdot \theta', \theta_1 \lessdot \theta'_1, \ldots, \theta_N \lessdot \theta'_N \,\}}$$

$$(\text{greaterFun}N{*})\ \ \frac{Eq \cup \{\, \texttt{Fun}N\texttt{*<}\theta, \theta'_1, \ldots, \theta'_N\texttt{>} \lessdot a \,\}}{Eq \cup \{\, a \doteq \texttt{Fun}N\texttt{*<}b', b_1, \ldots, b_N\texttt{>}, \theta \lessdot b', b_i \lessdot \theta'_i \,\}}\ \ b', b_i \text{ are fresh}$$

$$(\text{smallerFun}N{*})\ \ \frac{Eq \cup \{\, a \lessdot \texttt{Fun}N\texttt{*<}\theta', \theta_1, \ldots, \theta_N\texttt{>} \,\}}{Eq \cup \{\, a \doteq \texttt{Fun}N\texttt{*<}b, b'_1, \ldots, b'_N\texttt{>}, b \lessdot \theta', \theta_1 \lessdot b'_i \,\}}\ \ b', b_i \text{ are fresh}$$

Fig. 3. Extension of the type unification

The rule **reduceFun*N**** describes the reduction of the Fun*N** interfaces. This means that the parameters are in covariant respectively contravariant relations. The rules **greaterFun*N**** and **smallerFun*N**** describes the solutions of all greater respectively all smaller Fun*N**-types. This means that the parameters of the Fun*N**-types gets greater respectively smaller.

2.3 Example

In the following we show the functionality of the type inference algorithm **TI** by the application to the function `op` from Figure 2. First the function **TYPE** is called, that inserts type annotations, widely type variables as placeholders, and determines a set of type constraints. The abstract syntax of the program with type annotations inserted is:

```
op:$\underline{a_{op}}$ =
    ((m:$\underline{a_m}$) ->
       ((f:$\underline{a_f}$) -> f.apply(this:$\underline{\text{Matrix}}$, m:$\underline{a_m}$):$\underline{a_3}$)
       :$\underline{\text{Fun1*<}a_{app}, a_f\text{>}}$)
    :$\underline{\text{Fun1*<}a_{\lambda f}, a_m\text{>}}$
```

and the set of constraints is given as:

$$\begin{aligned}\{\, &(\texttt{Fun1}{*}\texttt{<}a_{\lambda f}, a_m\texttt{>} \lessdot a_{\texttt{op}}), (\texttt{Fun1}{*}\texttt{<}a_{app}, a_f\texttt{>} \lessdot a_{\lambda f}),\\ &(a_f \doteq \texttt{Fun2}{*}\texttt{<}a_3, a_1, a_2\texttt{>}), (\texttt{Matrix} \lessdot a_1), (a_m \lessdot a_2),\\ &(a_3 \lessdot a_{app}) \,\}\end{aligned}$$

With applying **greaterFun*N**** to $\texttt{Fun1}{*}\texttt{<}a_{\lambda f}, a_m\texttt{>} \lessdot a_{\texttt{op}})$ we get

$$\{\, (a_{\texttt{op}} \doteq \texttt{Fun1}{*}\texttt{<}b', b_1\texttt{>}), (a_{\lambda f} \lessdot b'), (b_1 \lessdot a_m) \,\}.$$

With applying **greaterFun*N**** to $\texttt{Fun1}{*}\texttt{<}a_{app}, a_f\texttt{>} \lessdot a_{\lambda f}$ we get

$$\{\, (a_{\lambda f} \doteq \texttt{Fun1}{*}\texttt{<}c', c_1\texttt{>}), (a_{app} \lessdot c'), (c_1 \lessdot a_f) \,\}.$$

With substituting $a_{\lambda f}$ in $a_{\lambda f} \lessdot b'$ and again applying **greaterFun*N**** we get

$$\{\, (b' \doteq \texttt{Fun1}{*}\texttt{<}d', d_1\texttt{>}), (c' \lessdot d'), (d_1 \lessdot c_1) \,\}$$

With substituting a_f in $c_1 \lessdot a_f$ and applying **smallerFun*N**** we get

$$\{\, (c_1 \doteq \texttt{Fun2*<}x, x'_1, x'_2\texttt{>}), (x \lessdot a_3), (a_1 \lessdot x'_1), (a_2 \lessdot x'_2) \,\}$$

With substituting c_1 in $d_1 \lessdot c_1$ and applying **smallerFun*N**** again we get

$$\{\, (d_1 \doteq \texttt{Fun2*<}y, y'_1, y'_2\texttt{>}), (y \lessdot x), (x'_1 \lessdot y'_1), (x'_2 \lessdot y'_2) \,\}$$

This leads to the following set of constraints (considering only the relevant constraints):

$$\begin{array}{l}
\{\, \texttt{Matrix} \lessdot a_1 \lessdot x'_1 \lessdot y'_1 \\
\;\; b_1 \lessdot a_m \lessdot a_2 \lessdot x'_2 \lessdot y'_2, \\
\;\; y \lessdot x \lessdot a_3 \lessdot a_{app} \lessdot c' \lessdot d', \\
\;\; a_{\texttt{op}} \doteq \texttt{Fun1*<Fun1*<}d', \texttt{Fun2*<}y, y'_1, y'_2\texttt{>>}, b_1\texttt{>}, \\
\;\; a_{\lambda f} \doteq \texttt{Fun1*<}c', \texttt{Fun2*<}x, x'_1, x'_2\texttt{>>}, \\
\;\; a_f \doteq \texttt{Fun}\mathit{2}\texttt{*<}a_3, a_1, a_2\texttt{>} \,\}
\end{array}$$

The result of **SOLVE** (considering only the relevant constraints and solutions) is given as following set of pairs:

$$\begin{array}{l}
\{\, (\{\, \texttt{b}_1 \lessdot a_2 \lessdot x'_2 \lessdot y'_2, \\
\qquad y \lessdot x \lessdot a_3 \lessdot c' \lessdot d' \,\}, \\
\;\;\; \{\, a_{\texttt{op}} \doteq \texttt{Fun1*<Fun1*<}d', \texttt{Fun2*<}y, y'_1, y'_2\texttt{>>}, b_1\texttt{>}, \\
\;\;\;\; a_{\lambda f} \doteq \texttt{Fun1*<}c', \texttt{Fun2*<}x, x'_1, x'_2\texttt{>>}, \\
\;\;\;\; a_f \doteq \texttt{Fun}\mathit{2}\texttt{*<}a_3, a_1, a_2\texttt{>} \,\}) \\
\;\;\; \mid\;\; \texttt{Matrix} \leq^* a_1 \leq^* x'_1 \leq^* y'_1 \,\}
\end{array}$$

The result of **TI** is given as the application of the **SOLVE**'s results to the result program of **TYPE**. The result consists of a set of typings for `op`:

```
class Matrix extends Vector<Vector<Integer>> {

    <y2’, b1 extends y2’, d’, y extends d’>[2]
    Fun1*<Fun1*<d’, Fun2*<y,  X , y2’>>, b1>
        op = (m) -> (f) -> f.apply(this, m);
    ...
}
```

where `Matrix` $\leq^*$ X.

If we compare this result with the Java 8 program in Figure 2 we see that the types are more general: On the one hand argument and result types are type variables and on the other hand there are more than one principal results (`Matrix` $\leq^*$ X).

This example shows that the results of the type inference algorithm are not unique in general. The reason is, that the type unification algorithm has multiple results.

```
class OL {

    m(a) { return a + a; }
    m(a) { return a || a; }
}


class Main {

    main(a) {
        ol;
        ol = new OL();
        return ol.m(a);
    }
}
```

Fig. 4. Type inference in the presence of overloading

Let us consider another example. In Figure 4 we show how the type inference algorithm deals with overloading. The result of the type inference for the method `main` is:

```
{ X main(X a) {
      OL ol;
      ol = new OL();
      return ol.m(a); }| X ∈ { Integer, String, Long, Double, Boolean, Float } }
```

In this example the property of multiple results is induced by the overloading of the operators `+` and `||`, while in `Matrix` the property is induced by the property that the type unification has multiple results.

Upto now, we have had a simple but practical solution to resolve multiple results. We have had an eclipse plugin [Sta15] as user interface such that the user can select the desired solution.

In this paper we consider a new approach, that resolves multiple solutions by extending the Java type system by intersections of function types.

3 Intersection function types

In this section we extend the Java type system by introducing intersections of function types. In [Plü08] we considered this for Java without `Fun`N`*`–types. Now we extend the idea to function types.
Let us look again on the class `Matrix` from Section 2. A first approach to define an intersection type could be to introduce for each supertype of `Matrix` an element (Figure 5). This definition makes less sense, as there are many subtype relations

[2] The constraints are given here as bounded type variables for fields, which is permitted only in methods in Java

```
op : Fun1*<Fun1*<d′,Fun2*<y,Vector<? extends Vector<? extends Integer>>, y2′>>,b1>
   & Fun1*<Fun1*<d′,Fun2*<y,Vector<? super Vector<? super Integer>>, y2′>>,b1>
   & ... &
   & Fun1*<Fun1*<d′,Fun2*<y,Vector<Vector<Integer>>, y2′>>,b1>
   & Fun1*<Fun1*<d′,Fun2*<y,Matrix, y2′>>,b1>
```

Fig. 5. Intersection type of op

```
op : Fun1*<Fun1*<d′,Fun2*<y,Vector<? extends Vector<? extends Integer>>, y2′>>,b1>
   & Fun1*<Fun1*<d′,Fun2*<y,Vector<? extends Vector<? super Integer>>, y2′>>,b1>
   & Fun1*<Fun1*<d′,Fun2*<y,Vector<? super Vector<Integer>>, y2′>>,b1>
```

Fig. 6. Reduced intersection type of op

between elements of the intersection. Therefore a better approach would be to define the type of op as the intersection of all maximal elements in the subtyping ordering. Then the type of op would be as given in Figure 6.
In general a principal type should be defined. The idea of principal typing is, that if an expression has multiple types, there is one type, from which all other types are derivable. This type is called the principal type.
E.g. in [DM82] a principal type for functional programs is defined, where the possibility to derive is the generic instantiation of type variables. E.g. the identity function has the principal type `id: a -> a`, where `a` is a type variable. This means all other types of `id` are instantiations of `a -> a`, e.g. `id: int -> int` or `id: char -> char`.
In [vB93] a generalization of this definition is given, that replaces the generic instantiation by an arbitrary derive-function.

We define for Java-TX the following principal typing:

Definition 1 (Java-TX principal typing). *An intersection type with minimal number of elements of an expression is a* principal type*, if any (non-intersection) type of the expression is a subtype of a generic instance of one element of the intersection type and the call-graphs are identical.*

For the explanation of this definition we give three further examples. We extend the matrix example by introducing a parameter for `Matrix<E>` and an additional class `intMatrix`, that contains the method `mul` (cp. Figure 7).
The type of `op` applied in the method `main` is

```
Fun1*<Fun1*<intMatrix, Fun2*<intMatrix,intMatrix, intMatrix>>, intMatrix>.
```

The corresponding element of the principal intersection type is

```
Fun1*<Fun1*<d', Fun2*<y,Vector<? extends Vector<? extends E>>, y2'>>, b1>
```

```
class Matrix<E> extends Vector<Vector<E>> {

    op = (m) -> (f) -> f.apply(this, m);
}

class IntMatrix extends Matrix<Integer>

    mul = (m1, m2) -> { ... }


    public static void main(String[] args) {
       m1; m1 = new intMatrix(...);
       m2; m2 = new intMatrix(...);
       (m1.op.apply(m2)).apply(m1.mul);}
}
```

Fig. 7. Parametrized Matrix

Let us consider again the class `OL` in Figure 4. The principal type of `main` is:

$$\begin{aligned}\mathtt{main} : {} & \mathtt{Integer} \rightarrow \mathtt{Integer} \,\&\, \mathtt{String} \rightarrow \mathtt{String} \,\&\, \mathtt{Long} \rightarrow \mathtt{Long} \,\&\, \mathtt{Double} \rightarrow \mathtt{Double} \,\&\\ & \mathtt{Boolean} \rightarrow \mathtt{Boolean} \,\&\, \mathtt{Float} \rightarrow \mathtt{Float}\end{aligned}$$

Finally we give an example that shows why the call-graph must be considered. Let the class `Put` in Figure 8 be given.

```
class Put {
    <T> putElement(T ele, Vector<T> v) {
      v.addElement(ele);
    }

    <T> putElement(T ele, Stack<T> s) {
      s.push(ele);
    }

    main(ele, x) {
      putElement(ele, x);
    }
}
```

Fig. 8. The class `Put`

The principal type of `main` is:

$$\mathtt{main} : \mathtt{T} \times \mathtt{Vector\text{<}T\text{>}} \rightarrow \mathtt{void} \;\&\; \mathtt{T} \times \mathtt{Stack\text{<}T\text{>}} \rightarrow \mathtt{void}.$$

If the call-graph would not be considered, $\texttt{T} \times \texttt{Stack<T>} \rightarrow \texttt{void}$ would not belong to the principal type, as `Stack` is a subtype of `Vector`. But this type is necessary as `main` defines different functions on `Vector` and `Stack`.

If we compare the matrix example with the others, we recognize, that the matrix example uses the lambda expression representation for functions, while in `OL` and `Put` methods are used. The Java-TX type systems allows for both representations intersection types.

4 Conclusion and outlook

4.1 Conclusion

We have presented an extension of the Java type system. On the one hand we proposed to introduce real function types. We gave an approach similar to the approach in Scala. We showed how both concepts, the concept of using functional interfaces as target types for lambda expressions, as well as our concept of real function types, can be integrated. So the advantages of both concepts can be used.
We showed the necessary extension of our type inference algorithm to use real function types.
On the other hand we have introduced function intersection types, that are in general results of our type inference algorithm.

4.2 Outlook

For the implementation of both the real function types and the intersection types generics in byte-code are necessary. In [ORW00] two ways to compile PIZZA [OW97] (an early Java extension with generics) are given. Beside the common homogenous compilation (type-erasures) there is given an approach of heterogenous compilation, which preserves the type parameters. This approach is designed for JVM version < 5. This approach has to be redesigned and adopted to version 8.

References

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. *Proc. 9th Symposium on Principles of Programming Languages*, 1982.

[GJS+14] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java® Language Specification.* The Java series. Addison-Wesley, Java SE 8 edition, 2014.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java^{TM} Language Specification.* The Java series. Addison-Wesley, 3rd edition, 2005.

[Ode14] Martin Odersky. *The Scala Language Specification Version 2.9*, May 2014.

[ORW00] Martin Odersky, Enno Runne, and Philip Wadler. Two Ways to Bake Your Pizza – Translating Parameterised Types into Java. *Proceedings of a Dagstuhl Seminar, Springer Lecture Notes in Computer Science*, 1766:114–132, 2000.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, January 1997.

[Plü07] Martin Plümicke. Typeless Programming in Java 5.0 with Wildcards. In Vasco Amaral, Luís Veiga, Luís Marcelino, and H. Conrad Cunningham, editors, *5th International Conference on Principles and Practices of Programming in Java*, volume 272 of *ACM International Conference Proceeding Series*, pages 73–82, September 2007.

[Plü08] Martin Plümicke. Intersection Types in Java. In Luís Veiga, Vasco Amaral, Nigel Horspool, and Giacomo Cabri, editors, *6th International Conference on Principles and Practices of Programming in Java*, volume 347 of *ACM International Conference Proceeding Series*, pages 181–188, September 2008.

[Plü09] Martin Plümicke. Java type unification with wildcards. In Dietmar Seipel, Michael Hanus, and Armin Wolf, editors, *17th International Conference, INAP 2007, and 21st Workshop on Logic Programming, WLP 2007, Würzburg, Germany, October 4-6, 2007, Revised Selected Papers*, volume 5437 of *Lecture Notes in Artificial Intelligence*, pages 223–240. Springer-Verlag Heidelberg, 2009.

[Plü11] Martin Plümicke. Well-typings for Java$_\lambda$. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 91–100, New York, NY, USA, 2011. ACM.

[Plü14] Martin Plümicke. Functional Interfaces vs. Function Types in Java with Lambdas – Extended Abstract. In *Tagungsband der Arbeitstagung Programmiersprachen (ATPS 2014)*, volume Vol-1129, pages 146–147. CEUR Workshop Proceedings (CEUR-WS.org), 2014.

[Plü15] Martin Plümicke. More type inference in java 8. In Andrei Voronkov and Irina Virbitskaite, editors, *Perspectives of System Informatics - 9th International Ershov Informatics Conference, PSI 2014, St. Petersburg, Russia, June 24-27, 2014. Revised Selected Papers*, volume 8974 of *Lecture Notes in Computer Science*, pages 248–256. Springer, 2015.

[PT98] Benjamin C. Pierce and David N. Turner. Local type inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 252–265, 1998.

[Sta15] Andreas Stadelmeier. Java type inference as an Eclipse plugin. In *Proceedings of the Studierendenkonferenz Informatik SKILL 2015*, 2015. (to appear).

[vB93] Steffen van Bakel. Principal type schemes for the strict type assignment system. *Journal of Logic and Computing*, 3(6):643–670, 1993.

Geschäftsprozesse kompiliert — Wichtige Unterstützung für die Modellierung

Thomas M. Prinz[1], Raphaël Charrondière[2] und Wolfram Amme[1]

[1] Friedrich-Schiller-Universität Jena, 07743 Jena, Deutschland
{Thomas.Prinz,Wolfram.Amme}@uni-jena.de
[2] École Normale Supérieure de Lyon, 69007 Lyon, France
Raphael.Charrondiere@ens-lyon.fr

Zusammenfassung Workflows — ausführbare (technische) Geschäftsprozesse — ähneln in ihrer Struktur und Funktionalität gerade im Umfeld der serviceorientierten Systeme einer modernen Programmiersprache. Der Kontrollfluss wird jedoch explizit mit Parallelität und Verzweigungsstrukturen als Graph modelliert. Die aus diesen Graphen entspringenden Programme sind dadurch selten strukturiert.
Für die Validierung und Übersetzung in ein interpretierbares und sicheres Übertragungsformat haben wir einen Compiler für Workflows entwickelt, der einen Modellierer direkt während der Entwicklung des Prozesses mit nützlichen Informationen unterstützt. Eine solche Information ist beispielsweise das Anzeigen aller möglichen Verklemmungen. Weiterhin kann der Compiler das Verhalten komplexer sogenannter inklusiver zusammenführender Gateways (OR-Joins) dem Nutzer erklären. Ein weiteres Novum ist die Benachrichtigung über mögliche Wettkampfsbedingungen und die damit zusammenhängende Transformation von unstrukturierten Programmen in die Concurrent Static Single Assignment Form.
In unserem Beitrag wird der Compiler *mojo* und seine verschiedenen Phasen vom eingehenden Geschäftsprozess bis hin zur Ausgabe vorgestellt. Im Zuge dessen gehen wir kurz auf die genutzten Analyse- und Transformationstechniken ein. Es handelt sich dabei um den ersten vollständigen Compiler für Geschäftsprozesse.

Schlüsselwörter: Compiler, Geschäftsprozess, Analyse, Verifikation

1 Einleitung

Große Softwareanwendungen werden heute im Hinblick auf Flexibilität, Skalierbarkeit und Übersichtlichkeit mit dem Pattern der *serviceorientierten Architekturen* (SOA) entwickelt. In einer SOA bieten Module ihre Dienste (Services) über Schnittstellen an. Solch ein Dienst steht dabei für die Durchführung einer (nach außen) abgeschlossenen Aufgabe. Eine Softwareanwendung, die auf dem SOA-Pattern basiert, nutzt eine Vielzahl solcher in sich abgeschlossener Dienste, um den Anwendungszweck, das Ziel, der Software zu erreichen. Wichtig ist also nicht mehr, *Wie* etwas getan wird, sondern *Was.* Sprich das *Geschäft*, das durch die

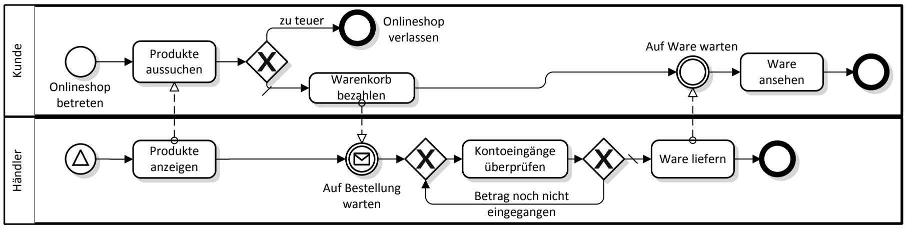


Abbildung 1. Vereinfachter Ablauf eines Einkaufs im Onlineshop

Verknüpfung mehrerer Dienste in einem Ablaufplan (*Prozess*) erledigt wird — beschrieben durch einen *Geschäftsprozess*.

Geschäftsprozesse verbinden verschiedene *Aufgaben* (Tasks) durch eine geordnete Reihenfolge (Prozess) zur Erfüllung eines Ziels (Geschäft). Abbildung 1 zeigt einen solchen Geschäftsprozess, der den (vereinfachten) Ablauf eines Einkaufs in einem Onlineshop repräsentiert und das Ziel hat, dass ein Kunde Produkte kauft, bezahlt und erhält. Die hier gewählte Notation für den Geschäftsprozess ist die *Business Process Model and Notation* (BPMN) 2.0 [1]. Der in dieser Sprache verfasste Prozess zeigt zwei involvierte Parteien: den Kunde und den Händler. Der Kunde setzt den Prozess in Gang, wenn er den Onlineshop betritt. Dies wird durch das Startevent (Kreis mit dünner Linie) gekennzeichnet. Der Teilprozess des Händlers startet ebenfalls mit dem Betreten des Onlineshops des Kunden, da er per Signal an diesen gebunden ist (Kreis mit Dreieck). Daraufhin zeigt der Händler dem Kunden seine Produkte. Daraus sucht sich der Kunde seine Produkte aus und entscheidet sich danach (Diamant mit einem X), entweder ob sie zu teuer sind oder ob er diese kauft. Sind die ausgewählten Produkte zu teuer, verlässt er den Onlineshop (Kreis mit dicker Linie) und sein Teilprozess ist beendet. Möchte er die Ware bezahlen, so geht er aus Sicht des Prozesses in die nächste Aufgabe. Ist der Kunde mit der Bezahlung fertig, so wird dem Händler eine Bestellung zugeschickt (Zwischenevent, doppelter Kreis mit Briefumschlag). Daraufhin kontrolliert der Händler den Geldeingang auf seinem Konto so lange, bis der Betrag eingegangen ist. Dann liefert er die Ware, auf die der Kunde bereits wartet. Der Teilprozess des Händlers ist damit abgeschlossen und der Kunde betrachtet abschließend noch seine erhaltene Ware.

Deutlich ist bei einem solchen Geschäftsprozess der Bezug zu Programmiersprachen zu sehen, mit zwei Unterschieden: *1)* ein Geschäftsprozess wird hauptsächlich durch eine relativ abstrakte, graphbasierte und unstrukturierte Modellierungssprache repräsentiert und *2)* besitzt einen anderen Sprachumfang.

Von diesem generellen Bezug zu Programmiersprachen ist es demnach erstrebenswert, wenn ein Geschäftsprozess ähnlich zu einem Programm in einer Programmiersprache ohne großen Aufwand automatisch *ausgeführt* werden kann. In diesem Zusammenhang wird dann über einen *Workflow*, also der technischen Umsetzung eines Geschäftsprozesses, gesprochen. Für einfache Geschäftsprozesse ist die Ausführung als Workflow derzeit sogar schon mit dem richtigen Werkzeug möglich (bspw. Activiti BPM Platform [2], Redhat jBPM [3] und IBM WebSphere [4]). Dabei wird der Prozess von einer *Workflowengine* interpretiert.

Ein Großteil von praxisnahen Prozessen kann jedoch nicht ohne weiteres ausgeführt bzw. als Workflow angesehen werden. Dies hat mehrere Gründe. Zum einen sind Beschreibungselemente moderner, meist graphbasierter Prozessmodellierungssprachen nicht vollständig spezifiert und deren Semantik könnte demnach unterschiedlich interpretiert werden. Dazu zählen beispielsweise die sogenannten inklusiven Gateways (OR-Split und OR-Join), die Fehlerbehandlung und Events. Zum anderen sind die existierenden Informationen im Prozess meist derart abstrakt, dass eine Übersetzung in maschinenverständliche Befehle nur durch einen Menschen vorgenommen werden kann.

Ein weiterer wichtiger Punkt ist die generelle Unterstützung des Prozessentwicklers bei der Modellierung eines Prozesses durch ein vernünftiges Werkzeug, dass dem Entwickler bereits während der Modellierung Fehler im Entwurf anzeigt. Teure Fehlverhalten sind ein wichtiger Knackpunkt zur Automatisierung. Tatsächlich gibt es bereits Werkzeuge (bspw. LoLA [5] und Woflan [6]), die in der Regel nach der Entwicklung Auskunft darüber geben, ob der Prozess frei von bestimmten Fehlern ist. Diese verwenden aber hauptsächlich Techniken zur Untersuchung des Zustandsraums, die in manchen Fällen einen exponentiell wachsenden Zustandsraum in Bezug auf die Größe des Prozesses generieren. Viel schwerwiegender ist jedoch, dass immer nur die *Fehlerwirkung*, sprich der Effekt, der durch einen Modellierungsfehler entsteht, untersucht wird. Aus Gründen der Berechnungskomplexität wird sogar nur die erste erreichbare Fehlerwirkung gesucht und danach die Fehlersuche abgebrochen. Die Information über eine solche Fehlerwirkung ist auch wichtig, kann aber nur unter Umständen zur Findung des *Fehlerzustands* bzw. der *Fehlerursache* beitragen. Dies wird in der Softwarequalitätssicherung auch als die *Entfernung* zwischen dem Fehlerzustand und der Fehlerwirkung bezeichnet. Ebenso ist auch die Entfernung zwischen der Fehlerwirkung von der eigentlichen *Fehlhandlung* (bspw. der falsche Einsatz eines Beschreibungselements) entsprechend hoch. Erschwerend kommt noch hinzu, dass sich zwei Fehler gegenseitig *maskieren* können, d.h., die Fehlerwirkung eines Fehlerzustands wird durch eine andere Fehlerwirkung eines Fehlerzustands korrigiert. Auch das *Blockieren* von weiteren Fehlerzuständen durch eine bereits aufgetretene Fehlerwirkung ist möglich — zum Beispiel bei einer Verklemmung, die alle nachfolgenden Fehlerzustände unerreichbar macht.

Abschließend zur Unterstützung bei der Modellierung finden außerdem Dateninformationen in geläufigen Techniken kaum bis keine Betrachtung und auch die Visualisierung und detailierte Beschreibung von gefundenen Fehlern (seien es Fehlerzustände oder -wirkungen) ist sehr unausgeprägt.

Insgesamt sehen wir den Bedarf der Evolution der Geschäftsprozesse, Prozessmodellierung und -ausführung hin zu einer (wenn auch abstrakteren) graphischen Programmiersprache mit einer integrierten Entwicklungsumgebung (IDE) bestehend aus einem Prozessdesigner, einem *Compiler* und der dazu passenden *virtuellen Maschine* (VM). Dafür zwingend erachten wir die folgenden Schritte: *1)* die Herleitung und Spezifikation einer relativ kompakten Kernsprache zur

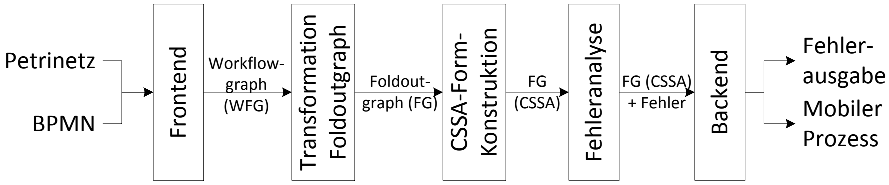


Abbildung 2. Phasen des Compilers

Prozessmodellierung mit häufig verwendeten Sprachelementen inklusive deren eindeutiger Semantik, *2)* die Herleitung und Spezifikation von Befehlen dieser Sprache zur Modifizierung von Informationen, *3)* die Definition von potentiellen Fehlerwirkungen und Fehlhandlungen, deren Fehlerursachen gefunden, angezeigt und detailliert beschrieben werden sollen, *4)* die Entwicklung von Anlaysen zur Findung von Fehlerursachen, *5)* die Entwicklung eines Compilers, der den Prozess einliest, nach Fehlerursachen und -wirkungen analysiert ggbf. Fehler anzeigt und beschreibt sowie in ein übertragbares und ausführbares Format übersetzt, und *6)* die Entwicklung einer virtuellen Maschine, die dieses Format einliest, auf seine Richtigkeit und Fehlerfreiheit überprüft und ausführt.

Ausführlichere Beschreibungen sind in einer vorangegangenen Arbeit [7] zu finden. In dieser stellten wir auch unsere Idee für den generellen Aufbau eines solchen Systems vor. Dabei wird das System in eine *Produzenten-* (der Compiler) und eine *Konsumentenseite* (die virtuelle Maschine bzw. Engine) unterteilt.

In dieser Arbeit stellen wir die bisherige Umsetzung der Produzentenseite in Form unseres Compilers für Geschäftsprozesse vor. Unter dem Namen *mojo*[1] entwickeln wir seit 2013 einen eigenständigen Compiler zur Analyse und Übersetzung von Prozessen. Er kann in existierende Prozessdesigner eingebunden werden, um deren Funktionalität zu erweitern (bisher im Activiti-Designer [2] getestet). Die Features unseres Compilers sind zum Beispiel, dass *1)* die Fehlerursachen aller potentiellen Verklemmungen innerhalb eines Prozesses gefunden werden, *2)* eine vollständige Semantik für inklusive zusammenführende Gateways (OR-Joins) genutzt wird und *3)* mögliche Wettkampfsbedingungen angezeigt werden können.

Im nächsten Abschnitt 2 untersuchen wir die unterschiedlichen Phasen des Compilers und zeigen dessen einzigartigen Funktionsumfang. Zum Schluss geben wir einen Ausblick auf zukünftige Arbeiten und Funktionalitäten in Abschnitt 3.

2 Aufbau des Compilers

Unser Compiler *mojo* orientiert sich an klassischen Phasen, in denen der Prozess immer wieder transformiert und analysiert wird. Das Phasenmodell in Abbildung 2 zeigt die derzeitigen Phasen von *mojo*.

[1] `http://sourceforge.net/projects/bpmojo/`, `http://www.bpmn-compiler.org`

Als Eingabe des Compilers dienen entweder ein in XML linearisierter BPMN-Prozess oder ein Prozess in Form der Petri Net Markup Language (PNML) [8]. Der Eingabeprozess wird dann im *Frontend* zunächst in die Zwischenrepräsentation des *erweiterten Workflowgraphen* [9] übersetzt. Erweiterte Workflowgraphen sind technische Repräsentationen des Prozesses als Graph. In der nächsten Phase (*Transformation Foldoutgraph*) wird der erweiterte Workflowgraph in den speziell für *mojo* entwickelten Foldoutgraphen übersetzt. Ein Foldoutgraph besteht in strukturierten Graphen immer nur aus einer Sequenz von hintereinander folgenden Knoten. Jeder dieser Knoten kann dabei jedoch in einen größeren Teilgraphen des Workflowgraphen entfaltet werden. Aus diesem Grund hat der Foldoutgraph auch seinen Namen erhalten. Der Gewinn dieser Transformation in den Foldoutgraph erlaubt schnelle Analysen dank einer impliziten Dominanz- und Postdominanzbeziehung zwischen den Knoten.

Nach der Konstruktion des Foldoutgraphen werden seine Instruktionen in die Concurrent Static Single Assignment Form (CSSA) [10] überführt (Phase *CSSA-Form-Konstruktion*). Im Zuge dieses Schritts muss der Compiler analysieren, welche Instruktionen parallel abgearbeitet werden und somit Wettkampfsbedingungen erzeugen können. Die gewonnene CSSA-Form der Instruktion macht es danach leichter dieses Wissen in Analysen zu verwenden. Diese Analysen werden dadurch wesentlich einfacher und effizienter.

Der eingehende Prozess befindet sich jetzt in der endgültigen Zwischenrepräsentation. Nun werden verschiedene *Fehleranalysen* durchgeführt. Aufgrund der Foldoutgraph-Struktur werden viele der Fehler bereits während der Transformation entdeckt. An dieser Stelle werden demnach noch die unstrukturierten Bestandteile des Foldoutgraphen analysiert und weitergehende Überprüfungen durch Prädikatenanalysen durchgeführt. Die gefundenen Fehler nutzt der Compiler für deren detaillierte Beschreibung zur Weitergabe an das entsprechende Modellierungswerkzeug (Phase *Backend*). Werden hingegen keine Fehler gefunden, linearisiert das Backend den Prozess und exportiert ihn. Er kann nun einfach von einer virtuellen Maschine ausgeführt werden.

Im weiteren Verlauf beleuchten wir die einzelnen Phasen etwas gründlicher.

2.1 Frontend

Wie bereits erwähnt wird im Frontend ein eingehender Prozess in einen erweiterten Workflowgraphen übersetzt. Der eingehende Prozess ist dabei entweder in der Notation eines BPMN-Prozesses notiert oder liegt als Petrinetz (PNML [8]) vor. Ausschnitt *a)* aus Abbildung 3 zeigt dies schematisch.

Für jedes Eingangsformat gibt es einen eigenen Parser, der zunächst die Syntax überprüft. Danach wird Knoten für Knoten und Kante für Kante des Eingangsprozesses in einen Workflowgraphen übersetzt [11]. Abbildung 4 zeigt die Übersetzung des Beispielprozesses aus Abbildung 1 in einen solchen Graphen.

Ein Workflowgraph ist im Wesentlichen eine allgemeinere und technischere Darstellungsform von Prozessen. Grundlegend stellt sie — ähnlich zu einem Kontrollflussgraphen — den Prozess als einen Graphen dar. Jedoch werden die Knoten je nach Funktionalität in verschiedene Typen unterteilt. Wir illustrieren

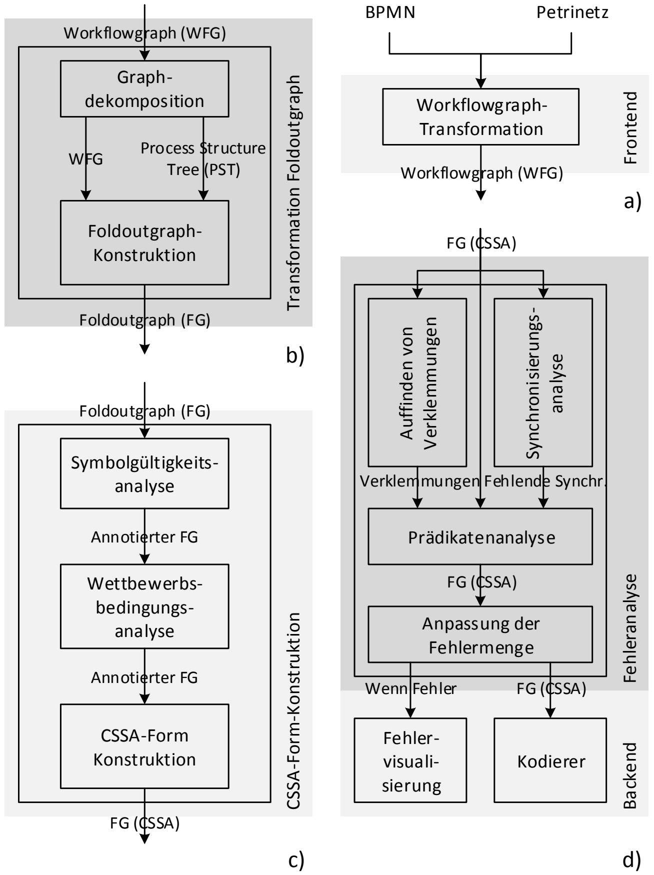


Abbildung 3. Detaillierte Phasen des Compilers

Startknoten als runde Kreise mit dünner Linie. Auf den ausgehenden Kanten dieser Knoten liegt zu Beginn der Kontrollfluss. Einfache Berechnungen, etc. werden in Aufgaben (Tasks, dargestellt als einfache Rechtecke) durchgeführt. Parallelität wird in AND-Forks erzeugt und in AND-Joins wieder zusammengefasst (schwarzgefüllte Rechtecke). Außerdem gibt es noch XOR-Forks und XOR-Joins für die Darstellung von Verzweigungen und Schleifen, im Graphen durch Diamanten dargestellt (XOR-Forks mit dünner, XOR-Joins mit dicker Linie). Analog dazu werden OR-Forks und OR-Joins dargestellt. Sie enthalten jedoch in der Mitte des Diamanten einen schwarzen Punkt. Ein OR-Fork und OR-Join ist in der Abbildung nicht zu sehen. Zu guter Letzt gibt es noch die Endknoten (Kreise mit dicker Linie), wobei in jedem Endknoten maximal ein Kontrollfluss endet.

Eine weitere Aufgabe des Frontends ist die Vereinigung der Start- und Endknoten zu einem einzigen Start- bzw. Endknoten unter Berücksichtigung der Semantik der Eingangssprache. Dies dient vor allen Dingen der wesentlichen

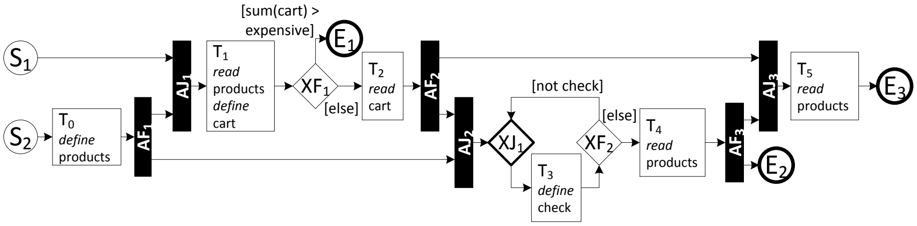


Abbildung 4. Der erweiterte Workflowgraph des Prozesses aus Abbildung 1

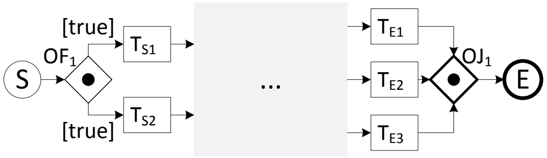


Abbildung 5. Verknüpfung mehrerer Start- und Endknoten zu einem einzigen

Vereinfachung von Analysen und Transformationen in den weiteren Phasen des Compilers. Vor unserer Einführung einer eindeutigen und vollständigen Semantik von OR-Joins [12], war das Verknüpfen verschiedener Endknoten zu einem einzigen ineffizient. Nun können (mit den passenden Bedingungen) die existierenden Start- und Endknoten jeweils durch einen Task ersetzt und mit Hilfe eines OR-Forks bzw. OR-Joins verknüpft werden, wie in Abbildung 5 zu sehen ist.

Außerdem transformiert das Frontend auch die Instruktionen des Prozesses. Die Erweiterung von Workflowgraphen mit Instruktionen wurde dabei erstmals in einer unserer früheren Arbeiten [9] genutzt. Sie werden *erweiterte Workflowgraphen* genannt. Im Zuge dieser Arbeit nutzen wir einfache Anweisungen, wie $define$ und *read*, um die Nutzung von Variablen anzudeuten.

2.2 Transformation Foldoutgraph

Der erweiterte Workflowgraph des Frontends wird an die nächste Phase des Compilers weitergereicht. Ab diesem Zeitpunkt sind die Analysen und Transformationen unabhängig von der gewählten Ausgangsnotation.

Der Compiler transformiert in dieser Phase den erweiterten Workflowgraphen in eine weitere Zwischenrepräsentation — den Foldoutgraphen. Foldoutgraphen zeichnen sich durch ihre Effizienz bei der Ausführung weiterer Transformationen und vor allen Dingen Analysen aus. Für strukturierte Prozesse besteht jeder Foldoutgraph aus einer Sequenz von Knoten — ohne Sprünge, Schleifen, Parallelität, etc. Dafür können bestimmte Knoten entfaltet werden. Diese Knoten haben dabei einen speziellen Knotentyp und stehen dabei beispielsweise für Verzweigungen, Schleifen und Parallelität.

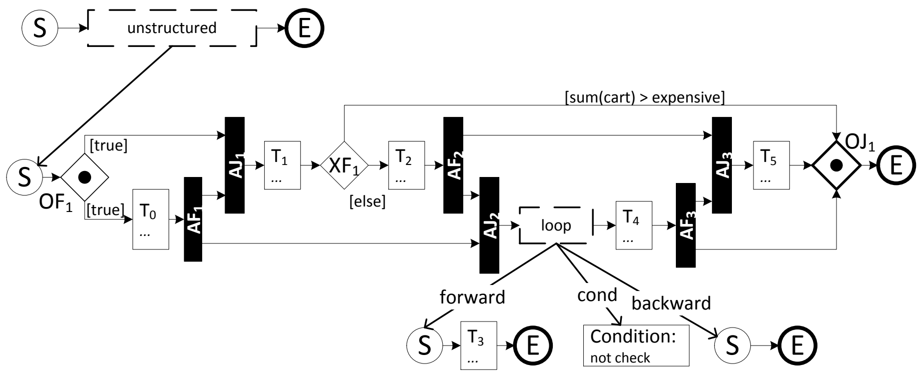


Abbildung 6. Der Foldoutgraph des Workflowgraphen aus Abbildung 4

Die Idee des Foldoutgraphen wurde von *OP2* [13], einem portablen Oberon-Compiler, und dem mobilen Code *SafeTSA* [14] inspiriert. Der Vorteil ist, dass der Dominator- und Postdominatorbaum und somit auch die Gültigkeit von Variablendefinitionen, etc. implizit gegeben sind. Außerdem ist das Auffinden vieler Modellierungsfehler schon während der Transformation möglich.

Da die meisten Workflowgraphen von realen Prozessen leider unstrukturiert sind, können Knoten des Foldoutgraphen auch zu unstrukturierten Teilgraphen entfaltet werden. Diese Teilgraphen folgen der Definition normaler erweiterter Workflowgraphen. Als Beispiel zeigt Abbildung 6 den Foldoutgraphen unseres Beispielprozesses. Da dieser stark unstrukturiert ist, sind lediglich die Schleife und das Fragment aus OR-Fork und OR-Join ausklappbar.

Die Transformation eines Workflowgraphen in einen Foldoutgraphen geschieht in zwei Schritten, wie in der schematischen Darstellung dieser Phase in Abbildung 3 *b)* zu sehen ist. Zunächst findet auf dem Workflowgraphen eine *Dekomposition* statt. Danach werden die aus der Dekomposition gewonnenen Daten genutzt, um den Foldoutgraphen zu konstruieren.

Die Dekomposition von Kontrollflussgraphen führten Johnson et al. [15] in ihrer Arbeit über den *Program Structure Tree* ein. Dabei werden *Single-Entry-Single-Exit*-Teilgraphen gesucht, wobei diese genau einen eingehenden und einen ausgehenden Knoten besitzen. Für unsere Dekomposition haben wir den Algorithmus von Vanhatalo et al. [16] übernommen, der von einem Prozess einen *Refined Process Structure Tree* (PST) ableitet. Dabei sucht er nach Regionen mit einer einzigen eingehenden und ausgehenden *Kante*.

Mit Hilfe des PST wird der Workflowgraph in linearer Zeit durchlaufen und die Fragmente des PST in Knoten und Kanten umgewandelt. Dabei wird auch bestimmt, welche Funktionalität das Fragment und somit der erzeugte Knoten widerspiegelt (Parallelität, Verzweigung, Schleife, usw.).

Während der Konstruktion des Foldoutgraphen wird außerdem überprüft, ob Fragmente korrekte öffnende und schließende Knoten haben. Beispielsweise wird geschaut, ob ein Fragment, dass mit einem AND-Fork beginnt auch immer mit einem AND- oder OR-Join abgeschlossen wird. Diese Fehler merkt sich der Compiler, repariert aber quasi den Graphen selbst für spätere Analysen.

2.3 Konstruktion der CSSA-Form

Nach der Konstruktion des Foldoutgraphen werden seine Instruktionen in der Phase der *CSSA-Form-Konstruktion* (Abbildung 3 *c)*) in die *Concurrent Static Single Assignment* (CSSA) Form [10] übertragen. In der CSSA-Form wird jede Variable (statisch) nur genau einmal definiert und bei jeder Änderung gibt es eine Neudefinition der Variablen. Da bei der Zusammenkunft zweier exklusiver Pfade mehrere Definitionen einer Variablen vorkommen können, wird an diese Stelle eine ϕ-Funktion für jede Variable eingefügt. Diese dient als virtuelle Kopieroperation, d.h., der Wert des ausgeführten Pfades wird durch die ϕ-Funktion gewählt.

In parallelen Programmen kann außerdem eine Variable gleichzeitig an zwei unterschiedlichen Stellen im Prozess modifiziert werden. Da dadurch Analysen fehlerhafte Ergebnisse liefern, wird vor jeder dieser möglichen, gleichzeitigen Zugriffe eine π-Funktion eingefügt. Diese speichert in einer neuen Kopie der Variablen den jeweilig zuletzt gefundenen Wert.

Um die CSSA-Form ableiten zu können, muss zunächst ermittelt werden, welche Variablen überhaupt an welchen Stellen im Programm Gültigkeit haben (Schritt *Symbolgültigkeitsanalyse*). Die Gültigkeitsbereiche von Variablen in explizit parallelen, strukturierten Programmiersprachen werden von der Struktur selbst vorgegeben. Da die Gültigkeitsbereiche in Workflows bisher jedoch noch keine Festlegung haben, benutzen wir den von uns in [17] vorgeschlagenen Ansatz: *Für einen Knoten n sind alle Variablen gültig, wenn diese in einem Dominator von n ebenfalls gültig sind oder dort definiert werden.*

Dieser Ansatz folgt den strukturierten Programmiersprachen, in denen die Struktur die Dominanzbeziehung vorgibt. Innerhalb des Foldoutgraphen können somit die Variablengültigkeiten in strukturierten Bereichen durch eine Rückwärtstraversierung für einen Knoten ermittelt werden. Für unstrukturierte Bereiche wird die Dominatorrelation abgeleitet.

Bezogen auf unser Beispiel aus den Abbildungen 4 und 6 bedeutet dies, dass die Variable *products* im Task T_1 nicht gültig ist. Dies kommt daher, dass der Task T_0 den Task T_1 nicht dominiert. Entsprechend sollte die Variablendefinition nach vorn gezogen werden.

Nun kann die Transformation in die CSSA-Form durchgeführt werden. Der Algorithmus zur Erzeugung der CSSA-Form von Lee et al. [10] leitet zunächst eine partielle Ordnung des Graphen ab, um festzustellen, welche Variablenzugriffe Konflikte verursachen können. Die tatsächliche partielle Ordnung zu bestimmen, ist Co-NP-vollständig. Wie jedoch bereits Lee et al. in ihrer Arbeit berichten, wird durch eine konservative Abschätzung lediglich die Anzahl der einzufügenden π-Funktionen erhöht. Dadurch werden mehr π-Funktion eingeführt als notwendig.

Anstelle der partiellen Ordnung ermittelt *mojo* Wettkampfsbedingungen. Die Analyse von Wettkampfsbedingungen in strukturierten Fragmenten des Foldoutgraphen ist einfach: Können zwei Zugriffe auf dieselbe Variable in zwei unterschiedlichen Knoten in Konflikt stehen, so müssen beide Knoten im Foldoutgraphen als ersten gemeinsamen Knoten eine Parallelität besitzen.

In unstrukturierten Bereichen (und somit für allgemeine Workflowgraphen) nutzen wir eine konservative Heuristik. Diese Heuristik besteht aus zwei Schritten für je zwei Zugriffe auf dieselbe Variable in zwei Knoten: *1)* Lösche von beiden Knoten die ausgehenden Kanten und verbinde beide durch ein XOR-Join (in einer Kopie des Graphen). *2)* Erreichen nun in einem Zustand zwei Kontrollflüsse das neue XOR-Join, so gibt es einen Konflikt.

Schritt *2)* der Heuristik bezieht sich auf die Fehlerwirkung einer *fehlenden Synchronisierung* — einem klassischen Fehler in der Prozessmodellierung. Wie in der folgenden Phase der Fehleranalyse beschrieben wird, haben wir einen effizienten Algorithmus entwickelt, der alle fehlenden Synchronisierungen in beliebigen Workflowgraphen findet [18].

Tests an einer Benchmark realer Geschäftsprozesse mit zufälligen Variablenzugriffen haben gezeigt, dass die Heuristik mit unserem Algorithmus zur Fehlererkennung gleiche Ergebnisse erzielt, wie die zeitaufwändige und im Zweifel exponentiell-große Zustandsraumerkundung.

Nachdem *mojo* die Wettkampfsbedingungen gefunden hat, werden diese zum einen zu deren späterer Visualisierung für den Entwickler gespeichert und zum anderen zur Platzierung der π-Funktionen genutzt. Danach werden die ϕ-Funktionen eingefügt. Die Konstruktion der CSSA-Form ist vollendet.

Wie in unserem Prozess aus Abbildung 4 gut zu sehen ist, besitzt unser Prozess keine solcher Konflikte, da die parallelen Prozesse quasi sequentiell verlaufen.

2.4 Fehleranalyse

Der ursprüngliche Prozess liegt nun in Form eines Foldoutgraphen in CSSA-Form vor. Diese (endgültige) Zwischenrepräsentation wird genutzt, um Modellierungsfehler innerhalb des Prozesses zu finden.

mojo findet verschiedene Arten von Modellierungsfehlern. Wie bereits erwähnt, analysiert der Compiler den Prozess bereits in jedem Übersetzungsschritt: *1)* syntaktische Fehler, *2)* korrekte Verwendung von öffnenden und schließenden Knoten von Fragmenten und *3)* undefinierte Variablen und Wettkampfsbedingungen.

In der Phase der *Fehleranalyse* sucht *mojo* zusätzlich nach *Verklemmungen* (Deadlocks), *fehlenden Synchronisierungen* und *nicht lebendigen Knoten.*

Verklemmungen und fehlende Synchronisierung Bei Verklemmungen und fehlenden Synchronisierungen handelt es sich um klassische Fehler während der Kontrollflussmodellierung in Geschäftsprozessen. Bei einer Verklemmung verbleibt der Prozess in ein und denselben Zustand und kann daher nicht ordnungsgemäß terminieren. Dies geschieht bei AND-Joins, die nicht auf jeder eingehenden Kante einen Kontrollfluss anliegen haben, und bei zwei OR-Joins, die sich gegenseitig blockieren. Die linke Seite von Abbildung 7 *a)* zeigt eine klassische Deadlocksituation, die durch das Zusammenführen zweier alternativer Pfade durch ein AND-Join herbeigeführt wird.

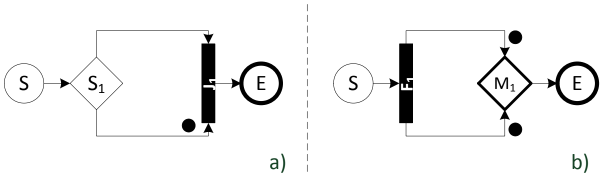


Abbildung 7. Eine Verklemmung *a)* und eine fehlende Synchronisierung *b)*

In einer Situtation mit einer fehlenden Synchronisierung besitzt eine Kante des Graphen zwei Kontrollflüsse (Tokens). Dies ist in Abbildung 7 auf der rechten Seite *b)* dargestellt. Diese Situation entsteht hier durch das Zusammenführen zweier paralleler Pfade durch ein XOR-Join.

Ursprünglich wurden die Begriffe Verklemmung (Deadlock) und fehlende Synchronisierung (Lack of Synchronization) von Sadiq und Orlowska eingeführt [19]. Seit dem entstanden die verschiedensten Techniken, um diese zu zeigen. Das Augenmerk dieser Techniken liegt auf dem Zustand, in dem die Fehlerwirkung zu Tage tritt. Die Fehler*ursache* wird nicht betrachtet. Außerdem wird nur die erste Fehlerwirkung gefunden und es wird nicht analysiert, ob nicht eine andere Fehlerwirkung bereits die Ursache für die gefundene Fehlerwirkung war.

In unseren Arbeiten zu diesem Thema [18, 20] haben wir einen neuen, compilerbasierten Ansatz entwickelt. Dieser erlaubt es bei kurzzeitiger Vernachlässigung der Dateninformationen (analog zu den bisherigen Techniken), alle *potentiellen* Fehlerursachen in beliebigen Prozessen zu finden und diese zu beschreiben. Potentielle Fehlerursachen müssen sich zur Laufzeit nicht durch eine Fehlerwirkung zeigen, da der eigentliche Kontrollfluss die Fehlerwirkung in manchen Fällen maskiert oder der Knoten, der die Fehlerursache bewirkt, gar nicht erreicht wird.

Die Algorithmen zur Analyse sind sehr effizient. Tests und Benchmarks haben gezeigt, dass dazu notwendige Analysen bereits im Hintergrund *während* der Modellierung durchgeführt werden und Ergebnisse liefern können.

Der wesentliche Ansatz, den wir verfolgen, ist das Ausfindigmachen von Knoten, deren Ausführung bestimmte Fehlerwirkungen hervorrufen können. Bspw. haben wir festgestellt, dass es auf allen Pfaden vom Startknoten zu einem AND-Join und auch von diesem AND-Join zu sich selbst Knoten geben muss, die *garantieren*, dass jede eingehende Kante des AND-Joins einen Kontrollfluss bekommt. Diese Knoten nennen wir *Aktivierungsknoten*. Die Abwesenheit eines solchen Knotens ergibt somit das Potential zu einer Verklemmung. Somit ist die Ursache einer aufgetretenen (und tatsächlichen) Verklemmung in einem AND-Join, das Fehlen eines solchen Aktivierungsknotens. Nehmen wir unser Beispiel aus Abbildung 6: Das AND-Join AJ_2 besitzt einen potentiellen Deadlock. Der Grund dafür liegt in dem XOR-Fork XF_1, das dafür sorgen kann, dass der Kontrollfluss nicht die obere eingehende Kante von AJ_2 erreicht, sondern über das OR-Join OJ_1 zum Endknoten verschwindet. Damit existiert auf keinem Pfad vom Startknoten zu AJ_2 ein Aktivierungsknoten. Bei der Untersuchung nach fehlenden Synchronisierungen verfolgen wir eine ähnliche Herangehensweise.

Die in unseren Vorarbeiten entwickelten Techniken wurden außerdem mit *mojo* weiterentwickelt, so dass auch Verklemmungen in OR-Joins erkannt werden können. Seitdem können auch Verklemmungen und fehlende Synchronisierungen erstmals in Prozessen gefunden werden, die OR-Joins beinhalten.

Prädikatenanalyse Unser Beispiel aus Abbildung 6 zeigt, dass es sich bei den gefundenen Verklemmungen und fehlenden Synchronisierungen um *konservative* Abschätzungen handelt: Wenn die Kosten für den Warenkorb ($sum(cart)$) immer kleiner oder gleich dem Wert *expensive* sind, dann tritt die gefundene Verklemmung nicht auf.

Diese Überabschätzung der tatsächlichen Fehler kommt durch die Vernachlässigung der Dateninformationen. Aus diesem Grund vollzieht *mojo* nach der Analyse von Deadlocks und Synchronisierungsfehlern eine *Prädikatenanalyse.* Dabei leitet der Compiler Zusicherungen ab. Solch eine Zusicherung $x = assert(y, predicate)$ steht für eine einfache Kopieroperation ($x = y$), garantiert jedoch zusätzlich, dass das Prädikat *predicate* für den Wert von y wahr ist. Dadurch werden (ähnlich zu ϕ- und π-Funktionen) zusätzliche Kopien einer Variable erzeugt.

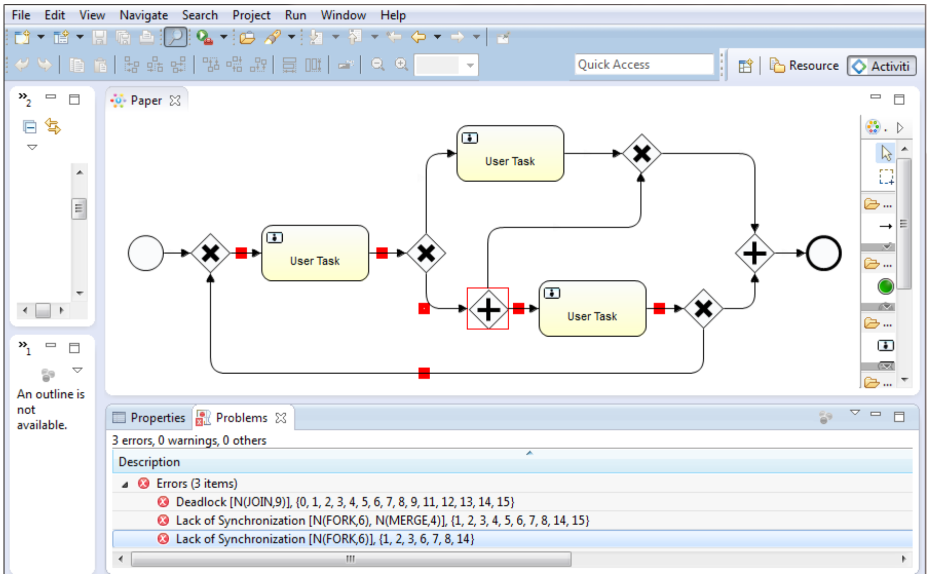
Die Prädikate erster Ordnung ergeben sich aus der Menge der Zuweisungen und Bedingungen. Gelten mehrere Prädikate für eine einzelne Instruktion, so werden diese konjunktiv zusammengeschlossen. Gibt es mehrere Definitionen auf exklusiven Pfaden für dieselbe Variable, so werden diese Prädikate disjunktiv verknüpft: Das Prädikat einer Variablen liegt in disjunktiver Normalform vor.

Der Vorteil der Zwischenrepräsentation und der Verwendung der CSSA-Form liegt in der einmaligen Definition von Variablen. Aus diesem Grund kann der Zustandsraum einer Variablen einmalig direkt bei seiner Definition annotiert werden. Durch die zusätzliche Einführung von Zusicherungen wird dies (ähnlich zur Static Single Information (SSI) Form [21]) auch nach Verzweigungen, etc. möglich, wo sich der tatsächliche Wert der Variablen nicht ändert, sondern nur dessen garantierter Zustandsraum. Zur Ableitung der Prädikate mittels Datenflussanalyse verweisen wir an dieser Stelle auf unsere Vorarbeiten [22].

Mit Hilfe der Prädikate überprüft *mojo*, ob bspw. ein als nicht aktivierend eingestufter Knoten eines AND-Joins unter den Zusicherungen doch ein Aktivierungsknoten ist. Diese Überprüfung findet unter Zuhilfenahme des SMT-Lösers *SMTInterpol* [23] statt.

Insgesamt kann die Abschätzung der Fehler mit Hilfe der Prädikatenanalyse präzisiert werden. Wir geben die eliminierten Fehler dennoch als Warnung aus, da sie nur durch den bedingten Kontrollfluss verhindert werden.

Nicht lebendige Knoten Die Prädikatenanalyse wird auch zur Warnung des Modellierers über *nicht lebendige Knoten* genutzt. Ein nicht lebendiger Knoten ist ein Knoten des Prozesses, der aufgrund der Bedingungen, die auf jedem Pfad zu ihm vorhanden sind, *niemals* ausgeführt werden kann. Dies ist vergleichbar mit *Deadcode Elimination.*



3 errors, 0 warnings, 0 others

Description

Errors (3 items)

Deadlock [N(JOIN,9)], {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, 14, 15}

Lack of Synchronization [N(FORK,6), N(MERGE,4)], {1, 2, 3, 4, 5, 6, 7, 8, 14, 15}

Lack of Synchronization [N(FORK,6)], {1, 2, 3, 6, 7, 8, 14}

Abbildung 8. Fehlerdiagnose im Activiti Designer [18]

2.5 Backend

Das Backend von *mojo* erfüllt zwei Aufgaben: Die Visualisierung von Fehlern im Prozessmodellierungswerkzeug und die Linearisierung des Foldoutgraphen bei festgestellter Korrektheit des Prozesses.

Das Backend ist abhängig von der Workflowengine, die den Prozess ausführen soll, als auch vom gewählten Modellierungstool. Zum Zeitpunkt dieser Arbeit wurde *mojo* in den *Activiti Designer* [2] integriert. Der Designer nutzt sowohl den Compiler in jedem Motivierungsschritt des Prozesses als auch die von dem Compiler zurückgegebenen Fehlerinformationen. Diese Fehlerinformationen werden dann in einem speziellen Fenster angezeigt und direkt im Prozess illustriert. In einem Fehlerdiagnosemodus können die Fehler im Einzelnen untersucht werden. Dabei werden viele Informationen über den Fehler, wie der verursachende Knoten, die Fehlerwirkung und eine Fehlerbeschreibung graphisch im Prozess hervorgehoben. Abbildung 8 zeigt eine Bildschirmaufnahme der Applikation. Eine solch detailierte Fehlerdiagnose und die vielen verschiedenen Arten von Fehlern sind in der Prozessmodellierung einzigartig.

Der Kodierer vollführt derzeit eine einfache Linearisierung des Foldoutgraphen aus Mangel einer entsprechend allgemeingültigen virtuellen Maschine (Prozessengine). An dieser Stelle sind Techniken, wie sie in dem SafeTSA-Format [14] verwendet werden, angedacht: Ein inhärent typ- und referenzsicheres und damit einfach zu verfizierendes mobiles Format. Damit wird sowohl die Verifikation und das Einlesen von Prozessen als auch die Ausführung beschleunigt.

3 Ausblick

In dieser Arbeit haben wir unseren Compiler *mojo* vorgestellt. Er ist der erste Compiler für (technische) Geschäftsprozesse. Er bietet zahlreiche Features, die in der Analyse und Transformation von Prozessen heute einzigartig ist: *1)* Vollständige Unterstützung von OR-Forks und OR-Joins, *2)* Schnell analysierbare Zwischenrepräsentation, *3)* Anzeigen von Wettkampfsbedingungen, *4)* Analyse von Gültigkeitsbereichen von Variablen, *5)* CSSA-Form, *6)* Vollständige Auflistung aller Verklemmungen und Synchronisierungsfehler, *7)* Einbeziehung von Dateninformationen, *8)* Analyse von toten Knoten und *9)* Detailierteste Fehlerdiagnostik mit Fehlerursache und Fehlerwirkung.

In zukünftigen Arbeiten soll *mojo* zur Produktreife für die Forschung weiterentwickelt werden. Dabei gehören auch weitere Analysen, wie das Auffinden von Zugriffen auf vorher gelöschte Variablen, Adaption der Techniken zur Kontrollflussentfaltung aus unseren früheren Arbeiten [22] und Korrekturvorschläge. Zudem soll die Veröffentlichung der neuen Techniken durch einen quelloffenen Download stattfinden.

Literatur

1. OMG: Business Process Model and Notation 2.0. formal/2011-01-03 (2011)
2. Alfresco: Activiti BPM Platform. `http://activiti.org/` (last access: February 18, 2015)
3. redhat: jBPM - Open Source Business Process Management - Process engine. `http://www.jbpm.org/` (last access: February 18, 2015)
4. IBM: IBM WebSphere software - United States. `http://www.ibm.com/software/websphere/` (last access: February 18, 2015)
5. Hinz, S., Schmidt, K., Stahl, C.: Transforming BPEL to petri nets. In van der Aalst, W.M.P., Benatallah, B., Casati, F., Curbera, F., eds.: Business Process Management, 3rd International Conference, BPM 2005, Nancy, France, September 5-8, 2005, Proceedings. Volume 3649. (2005) 220–235
6. Verbeek, H.M.W.E., van der Aalst, W.M.P.: Woflan 2.0: A petri-net-based workflow diagnosis tool. In: ICATPN. (2000) 475–484
7. Prinz, T.M., Heinze, T.S., Amme, W., Kretzschmar, J., Beckstein, C.: Towards a compiler for business processes - a research agenda. In de Barros, M., Rückemann, C.P., eds.: SERVICE COMPUTATION 2015: The Seventh International Conferences on Advanced Service Computing. Volume 6., Nice, France, IARIA Conference, ThinkMind Digital Library (March 22 2015) 49–54
8. Billington, J., Christensen, S., van Hee, K.M., Kindler, E., Kummer, O., Petrucci, L., Post, R., Stehno, C., Weber, M.: The petri net markup language: Concepts, technology, and tools. In: ICATPN. (2003) 483–505
9. Amme, W., Martens, A., Moser, S.: Advanced verification of distributed ws-bpel business processes incorporating cssa-based data flow analysis. International Journal of Business Process Integration and Management **4**(1) (2009) 47–59
10. Lee, J., Midkiff, S.P., Padua, D.A.: Concurrent static single assignment form and constant propagation for explicitly parallel programs. In Li, Z., Yew, P., Chatterjee, S., Huang, C., Sadayappan, P., Sehr, D.C., eds.: Languages and Compilers for Parallel Computing, 10th International Workshop, LCPC'97, Minneapolis, Minnesota,

USA, August 7-9, 1997, Proceedings. Volume 1366 of Lecture Notes in Computer Science., Springer (1997) 114–130

11. Spiess, N.: Realisation of a framework for the analysis of business processes (2013)
12. Prinz, T.M., Amme, W.: A complete and the most liberal semantics for converging or gateways in sound processes. Complex Systems Informatics and Modeling Quarterly, CSIMQ **Issue 4** (Oct 2015) [To be published].
13. Crelier, R.: OP2: A Portable Oberon Compiler. Technical Report ETH, Eidgenössische Technische Hochschule Zürich, Departement Informatik, Institut für Computer Systeme 125, Eidgenössische Technische Hochschule Zürich, Zurich, Switzerland (feb 1990)
14. Amme, W., von Ronne, J., Franz, M.: Ssa-based mobile code: Implementation and empirical evaluation. TACO **4**(2) (2007)
15. Johnson, R., Pearson, D., Pingali, K.: The program structure tree: Computing control regions in linear time. In Sarkar, V., Ryder, B.G., Soffa, M.L., eds.: Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20-24, 1994, ACM (1994) 171–185
16. Vanhatalo, J., Völzer, H., Koehler, J.: The refined process structure tree. In Dumas, M., Reichert, M., Shan, M., eds.: Business Process Management, 6th International Conference, BPM 2008, Milan, Italy, September 2-4, 2008. Proceedings. Volume 5240 of Lecture Notes in Computer Science., Springer (2008) 100–115
17. Prinz, T.M.: Proposals for a virtual machine for business processes. In Heinze, T.S., Prinz, T.M., eds.: Proceedings of the 7th Central European Workshop on Services and their Composition, ZEUS 2015, Jena, Germany, February 19-20, 2015. Volume 1360 of CEUR Workshop Proceedings., CEUR-WS.org (2015) 10–17
18. Prinz, T.M., Amme, W.: Practical compiler-based user support during the development of business processes. In Lomuscio, A., Nepal, S., Patrizi, F., Benatallah, B., Brandic, I., eds.: Service-Oriented Computing - ICSOC 2013 Workshops - CCSA, CSB, PASCEB, SWESE, WESOA, and PhD Symposium, Berlin, Germany, December 2-5, 2013. Revised Selected Papers. Volume 8377 of Lecture Notes in Computer Science., Springer (2013) 40–53
19. Sadiq, W., Orlowska, M.E.: Analyzing process models using graph reduction techniques. Inf. Syst. **25**(2) (2000) 117–134
20. Prinz, T.M., Spieß, N., Amme, W.: A first step towards a compiler for business processes. In Cohen, A., ed.: Compiler Construction - 23rd International Conference, CC 2014, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings. Volume 8409 of Lecture Notes in Computer Science., Springer (2014) 238–243
21. Ananian, C.S.: The static single information form. Master's thesis, Massachusetts Institute of Technology (MIT), Massachusetts (Sep 1999)
22. Heinze, T.S., Amme, W., Moser, S.: Compiling more precise petri net models for an improved verification of service implementations. In: 7th IEEE International Conference on Service-Oriented Computing and Applications, SOCA 2014, Matsue, Japan, November 17-19, 2014, IEEE Computer Society (2014) 25–32
23. Christ, J., Hoenicke, J., Nutz, A.: Smtinterpol: An interpolating SMT solver. In Donaldson, A.F., Parker, D., eds.: Model Checking Software - 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings. Volume 7385 of Lecture Notes in Computer Science., Springer (2012) 248–254

A Strategy for Generating Time-Predictable Code

Daniel Prokesch and Peter Puschner

Institute of Computer Engineering
Vienna University of Technology, Austria
{daniel,peter}@vmars.tuwien.ac.at

Abstract. *Prohibiting external control* is one of the key principles engineers apply when building time-predictable computer systems (e.g., time-triggered computer systems do not react to any external interrupts from sensors or devices, but all actions of these computer systems are triggered solely by the progression of the local clock). In this paper we apply this principle of *prohibiting external control* to code generation: The single-path code generator is a compiler that produces real-time code that does not contain any input-dependent control flow. All input-dependent control-flow dependencies are eliminated by if-conversion or by the generation of loops whose iteration counts are fixed. We explain the principle of operation of single-path code generation and illustrate how single-path code generation contributes to the time-predictable behavior of real-time computer systems.

Keywords: real-time computer systems, embedded systems, compilers, code generation, time predictability

1 Introduction

Real-time computer systems are used in safety-critical application domains like the automotive and aerospace domains. In these application domains computer systems must not only deliver functionally correct results. They must also produce these results at the right time. Otherwise a catastrophe like a plane crash might occur. Thus, an important property of each computer system used in a safety-critical real-time application is the temporal correctness of its operation. The worst-case analysis has to identify and analyze all worst-case timing scenarios such that the timely operation of the real-time computer system can be guaranteed for all phases of system operation.

Providing timing guarantees is a highly complex issue, especially as both the hardware and the software used in safety-critical real-time computer systems are getting more complex themselves. To keep systems nonetheless simple, the pre-planning design strategy for so-called time-triggered systems constructs a time schedule for all activities of the computer system at system design time [4, 3]. This means, the points in time when user tasks or operating-system tasks are started or when messages are sent is planned and written into scheduling tables before the system is started. At runtime, the system software of the real-time computer system interprets these tables as time progresses (time-triggered activation), thus strictly controlling the execution of all actions as planned. Such time-triggered systems do not allow for a dynamic change of the plan once the system is in operation, i.e., any external control over the sequencing of actions in the computer system is prohibited.

In this work we take the principle of prohibiting external control on actions in a real-time computer system one step further, from the scheduling level to the code level of single tasks. We present the *single-path code-generation strategy* that compiles C source code to machine code in such a way that the resulting machine code does not contain any input-dependent control flow. The absence of input-dependent control flow makes the execution of the machine code always take the same path through the instructions of the program and thus produces the same instruction trace every time the code is executed (therefore the name *single-path code* [10]).

Within this paper we will explain our approach to single-path code generation. We will first provide more motivation for using single-path code and then present the main idea behind single-path code generation (Section 2). We will then show how the LLVM compiler framework [5] can be extended with a single-path code generator (Section 3). We have run a number of experiments with the extended LLVM compiler framework. In these experiments, we compiled benchmark programs to single-path code and executed the compiled code on the Patmos time predictable processor [14]. These experiments and lessons learned are summarized in Section 4. Following this evaluation, we conclude the paper.

2 The Single-Path Approach

In this section, we would like to introduce the single-path code-generation strategy. In particular, we will answer the questions of (a) *why* one would like to generate and run single-path code and (b) *how* imperative code for real-time systems can be made to execute on the same instruction path for any inputs.

Above, we have motivated the use of single-path code by the fact that removing control-flow alternatives simplifies the worst-case execution-time (WCET) analysis of the generated code. In fact, the generation of single-path code eliminates the task of identifying (in)feasible program paths, one of the main subtasks of WCET analysis [9, 15], from WCET analysis. Besides, there are further advantages of using single-path code. The main advantages are summarized in the following paragraphs.

- The first and main advantage is that single-path code is much *easier to analyze* for its (worst-case) timing than traditional code – analyzing a single stream of instructions has a lower complexity than accounting for the timing of code that allows for a multitude of different instruction sequences.
- Second, if the instruction trace of a piece of code is always the same one can expect *smaller execution-time variations* than for code that executes

different instructions on each execution. This type of execution-time stability is advantageous for control software where a variable latency between inputs and outputs adversely affects control quality.

- Third, single-path code can be used to *thwart certain side-channel security attacks*: if all inputs are processed along the same instruction stream and with identical execution time (e.g., by using a processor with invariable instruction timing, [14]) then attackers cannot exploit observations of the code execution times to draw conclusions about the actual data being processed.
- Finally, precise knowledge about the instruction stream can be beneficial for *speeding up code execution*, e.g., by using the knowledge about the execution path to control the prefetching of code blocks into the fast levels of the memory hierarchy right before they are executed [2].

Making Code Execute on a Single Path

Traditional compilers generate code with input-data dependent branches in the control flow to realize input-data dependent code behavior. Such input-data dependent control flow realizes (a) the branching to conditional or alternative code of *if-then*, *if-then-else* or multi-way branches (e.g., *switch-case*) and (b) loop-exit branches for all types of *loop* statements. A single-path compiler, in contrast, must generate code that executes the same stream of instructions for all inputs. I.e., a single-path compiler has to provide code-generation patterns that bring forth data-invariant control flow for alternatives as well as loop constructs.

The single-path code generation uses the following strategies to generate code for alternatives respectively loop constructs:

Alternative constructs with input-dependent conditions are translated by means of *if-conversion* [1]: Instead of using conditional branches to achieve data-dependent code behavior, the single-path compiler generates predicated code [7], i.e., it serializes the code of input-dependent alternatives and uses predicates to control the activation of instructions and achieve the right code semantics at runtime.

Loops with input-data dependent exit conditions are translated into simple counting loops with a constant iteration count. Thereby, the iteration count of the generated loop is set to the maximum iteration count of the original loop[1]. The exit condition of the original source-code loop is used to compute a predicate for the execution of the loop body of the new loop that is itself translated into predicated code.

Further details about the single-path approach can be found in [10, 11]. The following part of the paper provides details about the realization of the single-path code generation in the LLVM compiler framework.

[1] We assume that the source code is is real-time code for which all loop bounds are known.

3 Generation of Single-Path Code

As a modern state-of-the-art compiler framework, LLVM operates in several phases. The frontend translates the source language to *bitcode*. Most optimizations are operating on this source language- and target-agnostic intermediate representation of LLVM. A backend translates the bitcode to target-specific machine instructions. Because the source code is not translated to machine code directly, the translation schemata described in [11] are not applicable directly. Where in the compilation process can the single-path code generation be integrated?

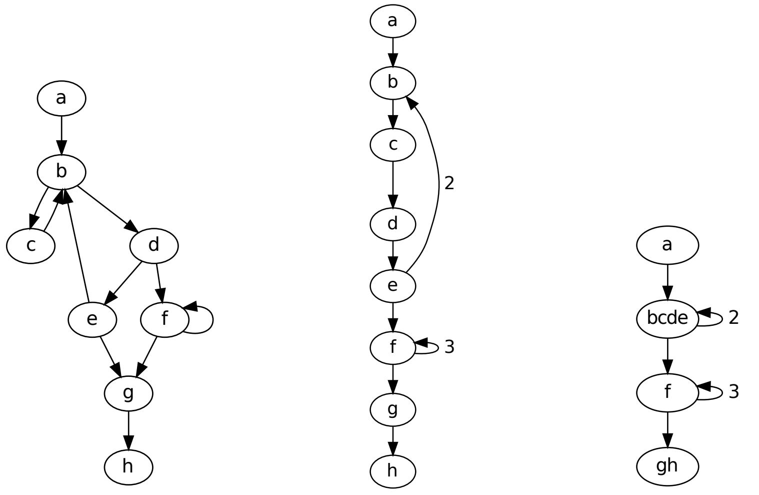
3.1 The Single-Path Graph Transformation

For the Patmos compiler, the single-path code generator is a set of program transformation passes that are executed late in the backend. As such, the problem of generating single-path code requires a suitable formulation on the program representation at that stage. To this end, we have developed the *single-path graph transformation* [8], that operates on the control-flow graph of a given function. Based on the algorithm of Park and Schlansker [7], it transforms the control-flow graph into a graph with linear structure, simple loops, and predicated nodes. It extends the algorithm [7] by transforming any reducible control-flow graph (not only the body of innermost loops) and by producing loops with constant iteration counts. Following the graph structure and the constraints regarding the loop back edges, there exists only a single path through the resulting graph. Furthermore, the transformation involves the insertion of instructions that control the value of the predicates assigned to the nodes.

Predicates are Boolean-valued variables that enable or disable operations. If the predicate is *true*, the operations are performed as usual, if the predicate is *false*, the operations have no effect. In terms of nodes in a control-flow graph, a predicate controls all the instructions of that node. Informally, the single-path graph transformation achieves the following:

> For every valid path in the original control-flow graph, the sequence of nodes on that path is equal to the sequence of nodes on the resulting graph with a predicate value of *true*.

The single-path graph transformation is best illustrated by an example. Figure 1a shows a control-flow graph before the single-path graph transformation. Figure 1b shows the single-path control-flow graph, with constant counts on the loop back-edges. Consider example paths π_1, π_2 in the original control-flow graph in the following table:



(a) Control flow graph (b) Single-path CFG (c) Simplified

Fig. 1: Example illustrating the original control-flow graph and the control-flow graph after the single-path graph transformation.

Original control flow graph	Single-path control flow graph
$\pi_1 = \texttt{abdebcbdegh}$	$\pi_1^{SP} = \underline{\texttt{ab}}\texttt{c}\underline{\texttt{debc}}\texttt{de}\underline{\texttt{b}}\texttt{c}\underline{\texttt{de}}\texttt{ffff}\underline{\texttt{gh}}$
$\pi_2 = \texttt{abcbcbdfffgh}$	$\pi_2^{SP} = \underline{\texttt{abc}}\texttt{de}\underline{\texttt{bc}}\texttt{de}\underline{\texttt{b}}\texttt{c}\underline{\texttt{d}}\texttt{e}\underline{\texttt{fff}}\texttt{f}\underline{\texttt{gh}}$

In the corresponding paths in the single-path control flow graph, π_1^{SP} and π_2^{SP}, respectively, the nodes with predicate value of *true* are underlined and their sequence equals the nodes of the original path. The singleton execution path always contains the same sequence of nodes, albeit with different predicate values.

Details on the single-path graph transformation, including how the predicates are assigned and changed along the execution path, can be found in [8].

3.2 The Patmos Processor

Before elaborating on the compiler passes, we briefly describe the relevant characteristics of the Patmos processor [14] that make it a suitable target for single-path code. Time predictability is the key principle in the design of Patmos. The timing of the instructions of the fully predicated instruction set is independent from the operands (except for latencies originating from the memory hierarchy). Features like dynamic instruction reordering and dynamic branch prediction are avoided in favor of static alternatives. Delays in the in-order dual-issue pipeline are exposed at instruction set architectural level (branch delay slots, load delays). The memory hierarchy is organized as a split cache architecture [13]. The caches are either software managed or at least controllable to obtain a known state, e.g., by flushing the cache contents.

3.3 Compiler Passes for Single-Path Code

As mentioned before, single-path code generation is performed late in the backend. This is due to following reasons. First, LLVM bitcode is SSA-based and predication-oblivious. Although there is support for partial predication in the form of a `select` instruction, which creates a new value as one of two operands depending on a Boolean operand, this form of predication is not sufficient to deal with the difficulties arising from instructions with side effects. Computing the values for alternative paths and discarding the unnecessary ones is only an option when safe values are provided, for example, to memory accesses and division operations in order to prevent access to invalid addresses and division by zero, respectively [6]. The machine instructions of the backend are predication-aware and have predicate operands. Second, the code structure is final at that stage and no instructions are inserted that could invalidate the single-path property. Calls to software arithmetic functions are already visible and the final number of required predicates is known. Third, we can perform optimized register allocation for predicate registers with detailed knowledge of the target, which we explain below.

The compiler passes for generation of single-path code are categorized as (i) preparatory passes, and (ii) the main transformation pass. The preparatory passes include a *unify return* pass, to guarantee that there is only one sink node in the control flow graph of each function, a *lower switch* pass to convert indirect jumps to a cascade of if-else statements, and *function cloning* to restrict single-path functions only to where they are required.

The main transformation pass performs the single-path graph transformation as described in the previous section. After computing predicates for each node in the graph, a specialized predicate register allocator is invoked, which assigns machine registers to the virtual predicates, on basic block (= node) level. Space for spilling predicate registers is allocated on Patmos' stack cache, and the 1-bit registers are stored packed into machine words. Live-ranges of predicates are predominantly nested and cover whole inner loops. This observation is exploited in Patmos to obtain a new set of available predicate registers when a loop is entered by spilling the whole predicate register file, and restoring it when the loop is left. After the assignment of physical registers, the instructions of each block are predicated accordingly. Function calls are executed unconditionally, passing the predicate to the called function. Then, instructions for manipulation and for spilling and restoring of predicate registers are inserted.

Finally, the basic blocks of the transformed control flow-graph are merged wherever possible, as illustrated in Figure 1c. This removal of basic block boundaries leads to a simplified control-flow graph structure with large basic blocks, which gives the final instruction scheduler more opportunities to generate compact and efficient instruction schedules.

4 Experiments

Having a compiler at hand that produces single-path code, we were particularly interested in answering following questions:

- How does the generated single-path code perform in the worst case, compared to conventionally generated code?
- How do latencies caused by the memory hierarchy affect the execution time?

To obtain answers to these questions, we evaluated the single-path code generator on a benchmark based on a real-world application. The *debie1* benchmark is based in the on-board software of the DEBIE-1 satellite instrument for measuring impacts of small space debris or micro-meteoroids, developed by Space Systems Finland Ltd for Patria Aviation Oy.[2]

We generated both conventional code and single-path code for the main tasks of the benchmark. We executed the conventional code to measure the observable range of execution times and additionally applied static analysis. For the measurement, we used `pasim`, the cycle-accurate simulator for Patmos. Each task is executed at least several hundred times in a benchmark run.[3] We used `platin` for static WCET analysis, a toolkit which is part of the compilation tool chain for Patmos [12].

We performed the evaluation with two different hardware configurations:

1. Ideal memory (*ideal*) - Memory accesses do not entail any additional access latency.
2. Ideal data cache (*dcideal*) - Only accesses that go through the data cache do not entail any additional latency. Memory accessed via Patmos' stack cache (2 kB) and method cache (4 kB) exhibits actual memory access latencies.

This choice is motivated by the fact that the serialization of control flow alternatives leads to an increase of the path lengths through the tasks. Masking the impact of the memory hierarchy enables us to quantify this effect solely at the instruction level. By allowing memory access for instructions and call frames, we can evaluate the single-path code in the context of real memory latencies, while maintaining execution-time invariability: On the single execution path, functions are called unconditionally, and space for call frames is allocated on the stack cache for those functions. Hence, every execution has the same sequence

[2] The source code is available at http://www.tidorum.fi/debie1/debie1-e-free.zip

[3] To be more precise, the number of executions of a task is in the range between 394 and 17795.

Task	SP Functions	Predicates	Configuration	Measured	Static Analysis	Single-Path	Ratio
TC_InterruptService	1	55	ideal	[17, 157]	163	306	1.88
			dcideal	[70, 445]	459	696	1.52
TM_InterruptService	1	10	ideal	[27, 38]	47	68	1.45
			dcideal	[78, 132]	141	163	1.16
HandleHitTrigger	3	31	ideal	[44, 7502]	12586	13879	1.10
			dcideal	[106, 7890]	13245	14351	1.08
HandleTelecommand	7	311	ideal	[67, 994]	994	3013	3.03
			dcideal	[179, 1294]	1294	5590	4.32
HandleAcquisition	17	234	ideal	[68, 26878]	29332	35695	1.21
			dcideal	[176, 29985]	36106	39824	1.10

Table 1: Results for the debie1 benchmark.

of accesses to the caches. In addition, we clear the caches before each entry to a single path function to obtain a well defined cache state. As a result, the generated single-path code has a singleton execution time by construction.

4.1 Results

The results of our experiments are shown in Table 1. The column “SP Functions” shows the number of functions that are involved in the tasks’ execution and require transformation. This contains the entry function of the task itself and all functions reachable in the call-graph. Column “Predicates” shows the total number of predicates required for the single-path version of the task. This number gives a hint about the breadth of the involved control-flow graphs. The column “Measured” shows the interval [min, max] containing the observed execution times of the conventional variants, while “Static Analysis” shows the WCET bound as computed by `platin`. The execution time of the single-path task code is given in column “Single-Path”. Because it is not known whether the actual worst-case path has been observed for the conventionally generated code, though we are primarily interested in worst-case guarantees, we have to consider the statically computed bound for a performance comparison with the single-path code. Column “Ratio” shows the execution time of the single-path code relative to the WCET bound of the conventionally generated code.

For these experiments, we can make some interesting observations. In all cases, the linearization of control flow alternatives leads to an increase in the execution time (bound), hence a ratio greater than 1. The highest ratio was obtained for HandleTelecommand (3.03 for *ideal*). In this particular task, the additional cost stems from serialization of a switch-statement: In the program, a message is read and processed accordingly depending on the message type. In the single-path variant, code for all the different cases is fetched and executed. The effect is even more pronounced when the code is loaded to the method cache (4.32 for *dcideal*). In the other tasks, the original control-flow structure has fewer alternatives in the control flow. As a result, the relative additional cost for loading code from main memory is lower, yielding a lower ratio in the *dcideal* case than in the *ideal* case.

4.2 Lessons learned

Our single path code generator is able to produce code without input-data dependent control flow. Targeting the time-predictable Patmos processor, this code generation strategy further results in code for real-time tasks that not only has a singleton execution path, but also is completely free from execution time jitter, making timing analysis trivial.

This property comes at a cost, as the experiments have shown. The execution time of the single-path code is higher than the statically computed worst-case execution time of the conventionally generated code. This is due to the serialization of control flow alternatives. One way to address the problem is to avoid input-data dependent control flow in the first place. But this has limited use, especially, when one has to deal with legacy code.

Another way would be to incorporate input-data dependence on a higher level of modeling. For example, the HandleTelecommand task of our benchmark performs different actions according to the message type of the incoming message. A type has its particular action, and the set of actions could be considered as *modes* of the task. There is little point in serializing all actions. Instead, by generating single-path code for each action individually, we would obtain a set of execution paths, where each path can be tied to the corresponding action.

5 Conclusion

Single-path code is free from input-data dependent control flow. Our single-path code generator is integrated in the compiler backend for the Patmos processor by adopting the single-path graph transformation. Patmos is a suitable target for single-path code, because it provides a predictable instruction set with full predication and software-controllable caches. Our experiments with the debie1 benchmark have shown that the generated single-path code is competitive with conventionally generated code in terms of worst-case performance, yet it is easier to analyze and exhibits stable execution-time behavior.

As future work we plan to implement compiler optimizations tailored to single-path code, for minimizing the cost introduced by control-flow serialization. Mode-specific single-path code will avoid complete serialization of input-dependent alternatives. It will provide a means to leave branches to mode-specific sections in the code, and the resulting mode-specific execution times could be used in a more differentiated way.

References

1. Allen, J., Kennedy, K., Porterfield, C., Warren, J.: Conversion of Control Dependence to Data Dependence. In: Proc. 10th ACM Symposium on Principles of Programming Languages. pp. 177–189 (Jan 1983)
2. Cilku, B., Prokesch, D., Puschner, P.: A time-predictable instruction-cache architecture that uses prefetching and cache locking. In: Proc. 18th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC) Workshops, 11th IEEE/IFIP International Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS). IEEE CS Press (2015)
3. Kopetz, H., Fohler, G., Grünsteidl, G., Kantz, H., Pospischil, G., Puschner, P., Reisinger, J., Schlatterbeck, R., Schütz, W., Vrchoticky, A., Zainlinger, R.: Real-time system development: The programming model of mars. In: Proc. IEEE International Symposium on Autonomous Decentralized Systems. pp. 190–199 (1993)
4. Kopetz, H., Zainlinger, R., Fohler, G., Kantz, H., Puschner, P., Schütz, W.: An engineering approach to hard real-time system design. In: Proc. 3rd European Software Engineering Conference. pp. 166–188 (1991)
5. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization (CGO'04). pp. 75–88. IEEE Computer Society (2004)
6. Mahlke, S., Hank, R., McCormick, J., August, D., Hwu, W.: A Comparison of Full and Partial Predicated Execution Support for ILP Processors. In: Proc. 22nd International Symposium on Computer Architecture. pp. 138–150 (Jun 1995)
7. Park, J.C., Schlansker, M.: On predicated execution. Tech. rep., Hewlett Packard Software and Systems Laboratory (May 1991)
8. Prokesch, D., Huber, B., Puschner, P.: Towards Automated Generation of Time-Predictable Code. In: Falk, H. (ed.) 14th International Workshop on Worst-Case Execution Time Analysis, WCET 2014, 2014, Madrid, Spain. OASIcs, vol. 39, pp. 103–112. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2014)
9. Puschner, P., Burns, A.: A review of worst-case execution-time analysis. Journal of Real-Time Systems 18(2/3), 115–128 (2000)
10. Puschner, P., Burns, A.: Writing temporally predictable code. In: Proc. 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems. pp. 85–91 (Jan 2002)
11. Puschner, P., Kirner, R., Huber, B., Prokesch, D.: Compiling for time predictability. In: Proc. SAFECOMP 2012 Workshops (LNCS 7613). pp. 382–391. Springer (2012)
12. Puschner, P., Prokesch, D., Huber, B., Knoop, J., Hepp, S., Gebhard, G.: The T-CREST approach of compiler and WCET-analysis integration. In: Proceedings of the 9th Workshop on Software Technologies for Future Embedded and Ubiquitious Systems (SEUS 2013) (2013)

13. Schoeberl, M.: Time-predictable cache organization. In: Proceedings of the 2009 Software Technologies for Future Dependable Distributed Systems. pp. 11–16. STFSSD '09, IEEE Computer Society, Washington, DC, USA (2009)
14. Schoeberl, M., Schleuniger, P., Puffitsch, W., Brandner, F., Probst, C.W., Karlsson, S., Thorn, T.: Towards a time-predictable dual-issue microprocessor: The Patmos approach. In: First Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011). pp. 11–20 (March 2011)
15. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The worst-case execution-time problem – overview of methods and survey of tools. ACM Trans. on Embedded Computing Systems 7(3) (2008)

Sharing Software Configuration via Specified Links and Transformation Rules

Markus Raab
markus.raab@complang.tuwien.ac.at

Institute of Computer Languages
Vienna University of Technology

Abstract

Configuration files are the dominant tool for local configuration management today. Up to now, applications cannot access any configuration file of their system, because they lack the knowledge where the configuration files reside, which syntax they use, and how a value is interpreted correctly. As a result, software systems are often poorly integrated. In this paper, we propose a specification for configuration files to mitigate these issues. Developers specify links and transformation rules to share configuration items between applications. We implemented tools and a library that integrate existing configuration files as well as a C/C++ code generator that makes sure that newly written applications are consistent with their specification. Our approach bridges across configuration file standards. It integrates configuration files we do not have control over. In a case study we demonstrate that our approach saves time, when unmodified applications need to be integrated into a coherent system. Additionally, we show that the run-time overhead of these links does not have significant impact on applications.

1 Introduction

Only a few factors determine how well-integrated an application, with respect to a software system, is: logging, external interfaces, user interfaces (such as dialogues), and user interaction (such as shortcuts and menus). In modern software these aspects are configurable. We only need to configure the software in a way that the system feels as if made from one piece. Currently, such endeavor is cumbersome in a heterogeneous system. Specific technologies only provide solutions within their respective field. Thus usually many technologies are involved, and a naïve approach often fails.

We faced this issue, next to many domain-specific ones, during an one-year project. The staff of the software engineering project varied between 3 and 5 software developers, who wrote about 50.000 lines of C and C++. The aim of the project was to engineer a platform for integrating different software applications from multiple customers. The platform enabled the modification of more than 200 configuration items which affect the behavior and features of the platform and the integrated applications.

Due to complexity of that specific system, caused by the domain-specific issues, we present our approach by the means of another issue instead: Nearly every graphical user interface (GUI) provides a shortcut for quitting the application. Also, nearly every application provides a way to use a different shortcut to change the default (that is often Ctrl+Q). Currently, no generic way exists that one application uses the shortcut of another application.

Submission to: 18. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS), Pörtschach, Austria, Oct-2015

We propose the following solution to give our applications access to configuration items of other applications: First, during installation of applications, we register their configuration files with information about their syntax. During startup of our application, we parse all necessary configuration files into key/value pairs with properties. A specification describes links and transformations between these abstract configuration items. Finally, we map the configuration items to variables of the programming language the application uses.

In the present paper, we will explain how a specification is used to drive the last two steps. In this specification, the developer defines which configuration items of other applications are used and how values are transformed. The specification is independent from a concrete programming language and extensible, e.g. for an editor:

```
[/our_editor/quit]
fallback/#0=/kde/kate/ActionProp/Default/file_quit
fallback/#1=/vim/map/:qa<CR>
fallback/#2=/emacs/keyboard-escape-quit
```

The first and the only required line of the specification defines the configuration item. We will call this unique id *key name*. The other lines of the section are *properties* of this configuration item. In this example, we introduced one property, called FALLBACK. Using this property we establish a link to another key name. It tells the system that, whenever the configuration item itself was not found, the value should be used from a fallback configuration item instead. We see that we add an array of such fallbacks, using the syntax #<*number*> for indexing.

The novelty of the approach is that it is transparent for any external tool in which order keys will be searched for. The specification is present at runtime and will be automatically enforced by a library. Additionally, code generation makes sure that applications use the configuration items in a type-safe way.

Our paper will answer the question: “Which properties in a specification are needed to share configuration items?”. This question is significant because it enables the developer to reuse configuration items in less time than building ad-hoc solutions for every single integration needed.

The paper is structured as follows: In Section 2 we explain the details of our approach and in Section 3 we describe Elektra, a framework that implements our approach. In Section 4 we evaluate and benchmark Elektra, then in Section 5 we compare our approach with other work and finally we conclude our paper in Section 6.

2 Elektra

2.1 Technology

Configuration files have a countless number of syntactic differences. E.g. lenses [1] are one way to describe syntax of configuration files. The repository of Augeas [7] (only covers some parts of Linux configuration files) already contains 181 lenses. Semi-structured data, e.g. JSON, YAML, XML and self-describing data [16], e.g. S-expressions and also JSON [3], are very popular for configuration.

To handle this diversity, as a first step, we transform such configuration files to an abstract syntax tree (AST). We use an AST that consists only of key/value pairs with properties. Using this abstraction we get rid of specialties in syntax. We use plugins as described in [9] that parse the configuration files and yield an AST. After the transformation to an AST, the nodes refer to key names and the properties store details that are necessary to reconstruct the configuration file. Let us consider a JSON file as example:

```
{
    "boolean_key": true
}
```

Elektra’s JSON plugin will transform this file to an AST. If we serialize the AST to the syntax we already used for the previous example we would get the following output:

```
[/boolean_key]
value=true
type=boolean
```

The example illustrates that self-describing data already has properties even without a specification. In a specification we directly link to `/boolean_key` regardless if a specification was written for it. Nothing novel so far, but using this technique we get an abstraction over the concrete syntax of configuration files. As a specialty, our approach handles specifications the same way as configuration files.

2.2 Resolve Sources

The location of the configuration's file name differs between different OS and distributions of the same OS. In our approach, the configuration file names from all applications are registered globally at installation time when we know the file name and its syntax. We use further configuration files to store this information. For every configuration file we add an OS-dependent plugin, called *resolver*, to handle the dynamic properties of the file name resolving. Depending on the OS context, resolvers will yield different file names.

Our approach introduces the abstraction that we will call *key database*. The key database is a tiny middleware between the plugins and the applications. Its main responsibility is the splitting and merging of the AST: plugins always get their part of the AST. The key database bootstraps itself whenever it is opened. It is library-based, that means the bootstrap process happens during the start of every application:

1. First the key database reads from a hard coded configuration file to know where the other configuration files are and which plugins (resolver, syntax of configuration file and others) should be used for each of them.

2. The resolvers determine the full configuration file names with information from the OS.

3. With the information from the previous steps, the key database builds up a data structure. In the data structure we lookup key names without any knowledge of the file name (and its syntax).

For example, consider that an application registers a JSON configuration file `app.j`, with the content as displayed above, in the key database at `/myapplication`. Then the full filename is the concatenation of a directory and `app.j` and the key name is `/myapplication/boolean_key`.

2.3 Namespace

Another dimension of configuration items is their *namespace*. Elektra supports following namespaces related to configuration files:

spec if the configuration file contains the specification.

dir if the configuration file is in a special directory (e.g. `.htaccess` of the apache web server).

user if the configuration file is in the user's home directory.

system if the configuration file is located system wide (e.g. below `/etc`).

In our approach, applications lookup all keys using the method `lookup` of a library. It has two arguments: the complete configuration `conf` (AST with all key/value pairs) and a key `key` without a namespace. No namespace is encoded in the application's source code. Instead, keys in different namespaces are considered for every lookup. The algorithm is straight-forward:

```
lookup(conf, key)
{
    s = lookupByKey(conf, spec / key);
    if (!s) return lookupBySpec(conf, s);

    ret = lookupByKey(conf, dir / key);
    if (ret) return ret;

    ret = lookupByKey(conf, user / key);
    if (ret) return ret;

    ret = lookupByKey(conf, system / key);
    if (ret) return ret;
    return 0;
}
```

First we search for the key, that contains the specification by using the `spec` namespace. If found, the internal method `lookupBySpec` is invoked with it. Otherwise, we do a cascading search of all namespaces. The operator `/` specifies that `key` is in the given namespace. The specification is only configuration represented by keys and defines how the lookup of keys work.

An abstraction of the configuration has been established. It enables us to uniquely identify configuration items. We do not have to care about the path to configuration files anymore.

2.4 Links

We already introduced the property array FALLBACK. It specifies which configuration items should be used as fallback when the configuration item itself was not found. The property array OVERRIDE complements the linking functionality. If this property is available, the configuration items linked will be preferred to the item itself. The property array NAMESPACE defines which namespaces should be considered and in which order. Finally, the property DEFAULT completes the linking functionality. It will be used if every configuration item mentioned was not found.

Given the AST `conf` and the key with properties `key` we define the search order by the following algorithm:

```
lookupBySpec(conf, key)
{
    for (number: 0 .. length(key, override)-1)
    {
        m = lookupByProperty(key, "override/#<number>");
        k = lookup(conf, m, withoutDefault);
        if (k) return k;
    }

    if (lookupByProperty(key, namespace))
    {
        for (number: 0 .. length(key, namespace)-1)
        {
            m = lookupByProperty(key, "namespace/#<number>");
            k = lookupByKey(conf, m / key);
            if (k) return k;
        }
    }
    else // if no property namespace exists
    {
        k = lookup(conf, key, withoutKey);
        if (k) return k;
    }

    for (number: 0 .. length(key, fallback)-1)
    {
        m = lookupByProperty(key, "fallback/#<number>");
        k = lookup(conf, m, true, withoutDefault);
        if (k) return k;
    }
    return lookupByProperty(key, "default");
}
```

The method `lookupBySpec` iterates over the three property arrays FALLBACK, NAMESPACE, and OVERRIDE. We see that the algorithm works recursively, but with special options for recursive invocations. The code shown here is not cluttered with those branches for clarity. If the property NAMESPACE is not specified (else branch), we use the cascading lookup as defined previously in the method `lookup`, but obviously do not search for the same key in `spec` again. The expression `m / key` means that key is in the namespace as stated by `m` and `"#<number>"` is the syntax for indexing. If neither an override, the key itself, nor any fallback was found, the algorithm returns the value as specified in the property DEFAULT.

For example, suppose we have no other configuration than the specification:

```
[/our_editor/quit]
namespace/#0=system
fallback/#0=/vim/quit
default="Ctrl+Q"

[/vim/quit]
namespace/#0=user
default=":q"
```

Then a call to `lookup` with `/our_editor/quit` as key will:

1. first lookup the key `/our_editor/quit` in the namespace `spec` successfully and call `lookupBySpec` with this key,

2. then skip override (because no property OVERRIDE is present),

3. then fail in searching for the key in the `system` namespace (because no configuration file is present),

4. then lookup the key `/vim/quit` in `spec` successfully and call `lookupBySpec` with this key,

5. skip override again,

6. then fail in searching for the key `/vim/quit` in the `user` namespace and
7. finally use the default value `Ctrl+Q`.

2.5 Value Transformation

The override/fallback mapping covers only the rare situation that the provider and consumer of the configuration item happens to interpret the same bit pattern in exactly the same way. In general, we need a unidirectional mapping of values from one bit pattern to another one if we want to reuse a configuration item. The straight forward way is to use a map:

```
[/our_editor/shortcut/quit]
transform=/kate/quit
transform/map/Ctrl+A=CTRL+A
transform/map/Ctrl+B=CTRL+B
transform/map/Ctrl+B=CTRL+C
... (23 more)
```

The property TRANSFORM is similar to override/fallback. On keys with this property, however, the key itself does not exist in a file. Instead its value is the result of a transformation specified by one of the properties TRANSFORM/TYPE, where "type" describes the type of the transformation. In the example, the transformation type is `map`. Such a mapping is practical for situations where an enumeration of all values is straight-forward, but cumbersome for others. Thus we use existing programming languages for sophisticated transformations:

```
[/our_editor/shortcut/quit]
transform=/kate/quit
transform/haskell=map toUpper
```

Which is much shorter than the previous example but has the same behaviour for valid input. In general, the transform code snippets must take one argument and return the transformed value. Obviously this specification language now is expressive enough for any transformation of values between applications, but we lack semantics when such a transformation should not happen or fails.

2.6 Skip Transformation

Sometimes the transformation is not possible or avoided on purpose:

- If the input value is not within the domain.
- If a runtime error (e.g. failed memory allocation) occurs.
- If someone decides that in the specific case it is better not to use the configuration item.

To support these semantics we use, depending on the programming language, either exceptions, errors, optional values or omit the return value. In such situations the configuration item will not be present for the application. The behavior is identical to a not-found key name.

For example, a typical request is that the application should only be adapted for an OS or desktop when the application is executed within the specific environment:

```
[/our_editor/shortcut/quit]
default=:qa!
transform=/kate/quit
transform/python=if kde_running():
    return value.upper()
```

In this example `our_editor` will use the quit shortcut from kate (which is part of KDE) only if KDE is the currently running desktop. Otherwise the key `/our_editor/shortcut/quit` will not exist.

2.7 Types and Code Generation

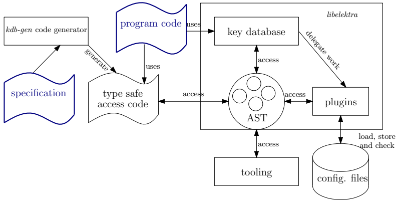
In our approach, every parameter has a type. If no property TYPE is given, an abstract top type will be assumed. This behavior guarantees that configuration items without the property TYPE still type-check safely [6].

Types give a better understanding how the parameter is used and provide a foundation to check if a concrete configuration is valid. In our approach we generate code for easy access to the configuration items in the same way as [12]. Then types become a necessity because only with types the compiler ensures that the usage of the configuration values is correct.

We conclude that every type used in the property TYPE must have an exact counterpart in the type systems of the target languages. CORBA IDL already defines a consistent mapping for many programming language and can be used with our approach.

3 Implementation

Figure 1: Architecture of Elektra



We call our framework Elektra. It consists of following parts:

1. A library (called *libelektra*) that transforms configuration files to an AST. It contains an implementation of the key database as discussed before.

2. We use a code generator (called *kdb-gen*) to ensure that the usage of the variables match the specification in statically typed programming languages. kdb-gen generates both C and C++ front ends and different documentation artifacts by using different templates.

3. The front end is a type safe access code that is generated by the specification.

The user of our approach has to implement only two parts (bold, blue boxes in Figure 1):

1. The specification with the properties FALLBACK, OVERRIDE, etc. (as discussed in Section 2).

2. The program code of the application consuming the configuration files has to be adapted to use Elektra (also called elektrify). The programmer must make sure that the program code uses the type safe access code and not directly configuration files nor environment variables so that the links will work.

We see in Figure 1 that the program code, that uses other configuration files, needs to use the key database in order to enable the consumption of other configuration files. Because of the plugins, however, other applications providing configuration files do not need to be modified. Elektra can use their configuration files directly.

3.1 Front end

The generated classes (C++) and functions (C) provide type safe and context aware access to configuration items [12] [10]. This layer is very thin. It is only responsible for looking up the configuration value and lexical casting the resulting string to a native data variable. It is straight forward to provide support for additional programming languages.

If desired, the front end can have a built-in copy of the specification. Using this technique, the application starts up without any configuration files.

In order to support the properties OVERRIDE and FALLBACK, we enhanced the lookup of values in the AST. Using the property TRANSFORM it will be transformed with the given rules. In rather static languages, e.g. C++, the transformation rule will be included in the generated code. In our C++ implementation we use policy-based design. Using policies programmers can modify the behavior of the front end. Additionally, it supports a separation of hand-written and generated parts.

E.g. the specification,

```
[/myapp/shortcut/quit_myapp]
default=CTRL+Q
type=string
transform=/kate/quit
transform/cpp=
    std::transform(value.begin(), value.end(),
    value.begin(), ::toupper);
    return value
```

will generate the following policy code (shortened for the sake of brevity):

```
class QuitMyappGetPolicy
{
public: typedef std::string type;
static type get(kdb::KeySet &ks, kdb::Key const& ) {
    type value = "CTRL+Q";
    kdb::Key found = ks.lookup("/kate/quit", 0);
    if (found) {
        value = found.get<std::string>();
        std::transform(value.begin(), value.end(),
        value.begin(), ::toupper);
        return value;
    }
    return value;
} };
```

The policy classes `GetPolicy` are responsible to lookup a configuration item in the case of a cache miss of the type safe access code [12]. The AST is denoted as `kdb::KeySet`. We see that the default value and the transformation are hard coded.

The generic tools we saw in Figure 1 cannot rely on code generation. Instead they read the specification dynamically. The implementation for both cases (code generated and dynamically) is a straightforward implementation of the pseudo-code as given in Section 2.

3.2 Key Database

The key database is a very thin layer that delegates all the work to Elektra's plugins. The plugins are responsible to resolve the configuration file name, as described in Section 2 and to parse and write configuration files. Different parsers are used for different formats:

1. JSON, INI and XML libraries handle semi-structured data.

2. Elektra's augeas [7] plugin handles most Linux configuration files, e.g. sshd, security/limits.conf.

3. Hand-written parsers handle other INI dialects and other configuration files, e.g. hosts, fstab.

The only responsibility left to the key database is to bootstrap the system and to pass the correct parts of the AST to the correct plugins.

4 Evaluation

We implemented the described artefacts and plugins in Elektra. Based on the experiences of the one-year project mentioned in Section 1 and later measurements, we will discuss the development time. The rest of the evaluation, however, is based on programs we wrote specifically for that purpose.

4.1 Development time

Measuring the overall time in a larger project was unfortunately not possible for us because using our approach individual configuration related tasks are done in a few minutes, which is difficult to track. We will only discuss development time of individual tasks. Additionally, we will show representative code to give a better understanding of necessary effort. The time measurement is based on the commits of our version control system.

The basic setup to use Elektra only consists of the following straight-forward lines:

```
#include <editor.hpp>

int main()
{
    using namespace kdb;
    KDB kdb;
    KeySet conf;
    Context c;
    Parameters par(conf,c);
    std::cout << par.myapp.shortcut.quitMyapp << "\n";
}
```

In the first line we include the code generated from the specification. After creating a handle to the key database, an AST for the configuration (called `KeySet`) and a `Context` [12], we finally create an instance of the generated class `Parameters`. Then we directly access configuration variables with the key name. The only difference to the specification is the usage of . instead of `/` to denote the key name.

The needed time to add one parameter is noteworthy: In only two minutes we were able to add a new configuration item that was fully documented and used in the application. The needed time does not change significantly if a small number of links exist. To add transformation keys, however, can take much longer, especially, if the original configuration item is not documented properly. To add command-line parsing ability for all parameters in a small application, we only needed six minutes (the template for generating the code already exists within Elektra).

We developed a large application (50.000 lines of C and C++) based on a specification describing configuration with Elektra. In that project we used several properties not described in this paper. Nevertheless, we experienced an improved development time and especially debugging time compared to another project with rather traditional means of configuration access: the direct use of a data structure along with a XML Schema Definition (XSD) validating configuration files written in XML. While one configuration change in our approach needed only a single change in the specification, up to more than 10 places needed to be modified in the project using XSD.

Adding new plugins to support new configuration file standards, unsurprisingly, takes significantly longer. Small tasks, e.g. integrating existing configuration parsers or writing a template similar to existing ones, were done by us within a day. For example, the INI plugin `ni`, that parses the syntax of the examples in this paper, has 158 lines of C code and was developed within a day (10:41:54 - 16:22:01). To support parsing properties following code in Elektra was needed:

```
Ni_node mcur = NULL;
while ((mcur = Ni_GetNextChild(current, mcur)) != NULL)
{
        keySetMeta (k, Ni_GetName(mcur, NULL), Ni_GetValue (mcur, NULL));
}
```

Explanations of the API and other details about development of the plugins are given in [9].

Adding new templates to support new programming languages or properties often takes a similar amount of time. For example, to add long option parsing support took less than one hour (10:53:30 - 11:33:04). The whole template that parses command line options has 261 lines of code. Templates are written in cheetah [15] with many utilities provided by Elektra, e.g. the fallback mechanism for C and C++ is included easily in new templates.

Here is a part of the template that implements the property FALLBACK:

```
@set $fa = $support.fallback(info)
@if len(fa) > 0
@for $f in $fa
    found = ks.lookup("$f", 0);
    if (found) {
        value = found.get<$support.typeof(info)>();
        $support.transform($info, $fa.index($f));
    }
@end for
@end if
```

Note that in cheetah template code (lines starting with `@`) is interwoven with C code. The array `fa` contains the properties FALLBACK. For every property FALLBACK the code for a `lookup` invocation and the value transformation is generated. The generated code from that template is shown in the earlier example `class QuitMyappGetPolicy`.

4.2 Links within a single specification

We implemented a word counting utility in C that internally relies on the links as described in this approach. The `wc` tool has the following features: it counts lines, words, chars, bytes and the length of the longest line. Without any option the tool will print lines, words and chars. With any configuration item given, it will only print the requested counters. Such interdependencies within configuration item are easily represented with the following links:

```
[/sw/wc/show/max_line_length]
type=boolean
default=false
opt=L
opt/long=max_line_length

[/sw/wc/show/no_default_args]
type=boolean
default=false
override/#0=/sw/wc/show/lines
override/#1=/sw/wc/show/words
override/#2=/sw/wc/show/chars
override/#3=/sw/wc/show/bytes
override/#4=/sw/wc/show/max_line_length
```

Using links avoids an implementation of the override/fallback logic in the application and has following advantages:

Changeability: Even when multiple applications use the specification item `no_default_args` we only have a single place to change the logic.

Independence: The links are available as data and can be used in any programming language or technology.

Traceability: The links exist explicitly and are traceable without program analysis.

Extensibility: Both the configuration items and the links can be extended with any desired property, e.g. `opt` in the example above permits us to generate commandline parameter parsing code that accepts `-L` and `--max_line_length`.

Performance: In a previous paper [12] we show that access of the configuration item has no overhead compared to access of native C++ variables.

4.3 Links between applications

We implemented a minimalist editor with configurable shortcuts in C++. The tool sloccount measured 3,106 total physical source lines of code. In this case study, we confirmed that other editors do not need any modification to share their configuration.

Elektra supports a large number of configuration file standards, including those mentioned in this paper (JSON, XML), those supported by Augeas (e.g. apache, ssh), some basic Linux configuration files (e.g. hosts) and various INI formats. To support a large number of standards is especially important for software integration between applications.

For vim and Emacs, however, none of these parser worked because their configuration file contains code. For such situations we implemented a plugin, called regexstore, that uses regular expressions ignoring all non-matches. In contrast to lenses regexstore only takes care of the parts of the configuration files we are interested in and does not understand the rest of the file. Given regexstore, we are able to integrate even vim and Emacs configuration files. Up to now, we did not find any configuration file we could not integrate and because plugins are written in universal programming languages we argue that any way to store configuration can be integrated into Elektra.

4.4 Benchmark Setup

We conducted the benchmarks on a hp® EliteBook 8570w using the CPU Intel® Core™ i7-3740QM @ 2.70GHz. The operating system is GNU/Linux Debian Wheezy 7.5. We used the gcc compiler Debian 4.7.2-5. We measured the time using `gettimeofday`. We executed each benchmark eleven times for the box-plots. The benchmark setup is identical to our previous benchmark [12].

We implemented a static and a dynamic variant of our algorithm:

In the dynamic variant keys contain the properties. That means that `lookupByProperty` of our algorithm is a dynamic lookup in a data structure. The application reads the specification in configuration files at runtime. In this variant, we have to lookup every property, even if they are not available.

In the static variant the override and fallback mechanism is compiled in the application. In this variant the code generator adds code for every link as specified. We already saw examples of this variant (the class and the template of `QuitMyappGetPolicy`). Only recursive links (not used in our benchmarks) are resolved in the same way as in the dynamic variant. That means also in this variant the specification needs to be read at runtime. Obviously, this approach has no overhead when no links are used.

4.5 Lookup Time

In the first benchmark we will measure the time needed for the lookup in the data structure. We generated 10 variables that have 0 to 9 properties OVERRIDE, e.g. the first and second configuration items:

```
[/benchmark/#0]
default=33
type=unsigned_long

[/benchmark/#1]
default=benchmark
type=unsigned_long
override/#0=/benchmark/override/#0
```

Using these configuration items we make 200,000 lookups. If we would only access the variable, we would not have any performance overhead (with any number of links) as described in [12]. In order to actually measure the lookup time, we synchronize the cache in every iteration. In Figure 2 we see the time grows linearly for a larger number of OVERRIDE-properties.

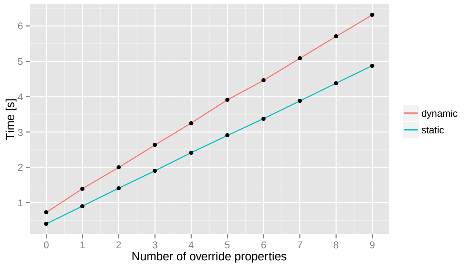


Figure 2: Lookup time in static and dynamic variant with linear scale for 200,000 lookups.

For the second benchmark measuring the dynamic variant we use the same specification and the same number of iterations. We see that we have a larger overhead because of the additional lookups required for every property. In the dynamic variant even not specified properties cause overhead.

Next to the constant difference of factor 1.8 the overhead also grows 22% faster in the dynamic variant because of additional property lookups needed. Another drawback of the dynamic variant is, that it does not allow compiled code to be used. Instead transformations need to be interpreted, again adding overhead. We conclude the static variant is faster, but it has a severe problem: applications directly using the key database without code generation (e.g. tooling) sees not transformed, i.e. incorrect, values. To answer the question if dynamic or static should be preferred, we have to answer the question if the lookup overhead is significant in an application.

4.6 Overall Runtime Overhead

Considering a full application, in our approach following steps are necessary:

kdbOpen: The bootstrapping as described in Section 2.

kdbGet: The reading of (other) configuration files.

process: The functionality of the application.

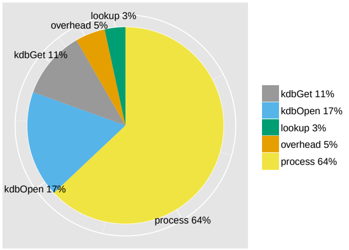


Figure 3: Full application using a static variant

Our setup is as follows. We benchmark the word counting tool already described earlier with the $\LaTeX$ file document of this paper (32KB size). We have a small configuration file that is read during startup. We profiled the application using Callgrind 3.7.0. Only two configuration files were involved: the bootstrapping configuration file (read during `kdbOpen`) and one configuration file for the application (read during `kdbGet`).

In Figure 3 we see that the processing of the file dominates with 64%. What obviously matters is the size of the configuration files. The transformation to an AST (using the ni plugin [9]) unfortunately is much slower than just counting the words (by a factor of 12 on files with the same size). Because of this `kdbOpen()` takes 17% and `kdbGet()` needs 11% of the overall cycles even with a small configuration files.

Without the links in the specification the number of lookups are reduced to 9 (from otherwise 27) cascading lookups (see the algorithm `lookup` in Section 2). Based on this, we know that the overhead of the links is only 5% in this application.

4.7 Threats to Validity

The main problem of our evaluation is that the applications used are rather small. The development times are only taken from a single project and need to be validated by additional case studies for external validity.

5 Related Work

Configuration management (CM) tools, e.g. [2] solve the problem stated in this paper differently. They copy each value to every place as needed. This approach, however, fails to work, when the user modifies configuration locally. Most computer systems today, however, are configured locally: mobile devices, laptops, tablets and non-business desktop systems. CM tools will not work when no distribution system between the nodes exists.

Reuse of software components facilitates reuse of software configuration. Modern desktop environments have a tight integration based on that principle. In this approach every software component is responsible for its own configuration. These components enable programs to modify settings in one place and taking effect for the whole system. Zdun [17] even argues that the concern "behavioral composition and configuration" should be treated as a first-class entity. While this approach has many advantages, its application is often difficult (e.g. programming language barriers) or even not possible (e.g. because of licensing issues).

Using lenses [1] as implementations allows us to quickly cover a lot of different configuration file formats, but lenses seem to fail [7] when we need complete abstraction from the concrete syntax of the configuration file. Type-safe lenses are only based on regular expressions and the resulting AST is very similar to the configuration file syntax and its internal order. In our approach we do not have such a limitation because we can transform keys to the desired structure.

Ontologies are used for sharing data. Gruber [5] describes general design criteria so that every specification has a minimal ontological commitment, e.g. we should tolerate that one date is "1993" and the other "March

1993". Enforcing a canonical representation would be an encoding bias. In our approach we solve this problem by transformations. Gruber also introduces references to uniquely identify publications. In our approach key names cover this concept.

XPointer [4] permits us to create links within XML documents. The main difference is that they are heavily dependent on XML technology. So they cannot be used for configuration files, where XML is not dominant. XPointer is not able to achieve the same as the properties OVERRIDE, FALLBACK and DEFAULT we described in this paper. XInclude [8] is also tied to XML technology, but different from XPointer, XInclude has an element fallback that is similar to the property DEFAULT as described in this paper. For the other properties we described, no equivalents exist.

Context-oriented programming [12] [10] is supplementing the approach described in this paper. It allows applications to be aware in which context they are, but does not enable them to be aware of other applications.

We are positive that possible extensions of our approach also improve safety [11].

A different type of configuration links are explained in [13] and formally developed in [14]. Similarities to our approaches are advantages regarding specification evolution, and the potential usage for internal fallbacks as we discussed in the evaluation. They are different because they:

- only refer from target to source specifications while Elektra supports references within specifications and directly to configuration items,
- are evaluated during generation of configuration and therefore cannot be as flexible as Elektra's run-time evaluation,
- only provide propositional logic to determine selection while Elektra facilitates programming languages to determine if transformation should be skipped, and
- seem not to support transformation rules which further limits their use.

6 Conclusion and Further Work

This paper describes a novel way to establish links between configuration items. We use data as a specification to define abstraction and to describe access on data. This specification alone suffices to share configuration, and even saves time in the process. As the specification is just data we can easily extend the approach to other programming languages. Nevertheless, the specification is powerful enough to allow applications to use any configuration item of any configuration file. Our approach avoids the introduction of a new configuration file format. Instead, existing configuration file formats are integrated using a global abstract syntax tree.

In the benchmark we compared a static and a dynamic implementation of our approach. The lookup time is not significant in either way, and the dynamic implementation has the advantage that it also works with tooling that does not use the code generator. So we prefer the dynamic variant.

To answer our research question: Three properties, namely FALLBACK, OVERRIDE and TRANSFORM are needed to share configuration between applications. Additionally, these links are also useful to implement fallback and override logic within a single program. In general, using the specification is superior to hard coding logic in the application because:

- External tools use the specification and thus present the same configuration as the application.
- The specification is enforced for every application accessing the configuration.
- It is transparent to the administrator which configuration values are to be preferred.

Currently, types need to be annotated manually for every key. As further work we plan to reconstruct the types using the links. By reconstructing the types of the transformation rules (e.g. when Haskell and ML are used), we will even infer transformed types. Furthermore, other transformations are waiting to be explored and evaluated. Last, but not least, we want to evaluate techniques which allow our approach to be used for unmodified binaries, e.g. by preloading `getenv()`.

Our contributions are:

- Describing an approach in which applications are aware of other applications' configurations, leveraging easy application integration.

- Implementing our approach for popular semi-structured data formats and Linux configuration files, downloadable from `http://www.libelektra.org`.
- Comparing a static and dynamic variant of the implementation.
- Providing experimental validation using a case study of significant complexity and evaluate performance.

References

[1] Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: resourceful lenses for string data. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 407–419, NY, USA, 2008. ACM.

[2] Mark Burgess et al. Cfengine: a site configuration engine. *USENIX Computing systems*, 8(3):309–402, 1995.

[3] Douglas Crockford. Json: The fat-free alternative to xml. In *Proceedings of XML*, volume 2006, 2006.

[4] Paul Grosso, Eve Maler, Jonathan Marsh, and Norman Walsh. Xpointer framework. *W3c recommendation*, 25, 2003.

[5] Thomas R. Gruber. Toward principles for the design of ontologies used for knowledge sharing. *International Journal of Human-Computer Studies*, 43(5–6):907 – 928, 1995.

[6] Niklaus Haldiman, Marcus Denker, and Oscar Nierstrasz. Practical, pluggable types for a dynamic language. *Computer Languages, Systems & Structures*, 35(1):48–62, 2009.

[7] David Lutterkort. Augeas–a configuration API. In *Linux Symposium, Ottawa, ON*, pages 47–56, 2008.

[8] Jonathan Marsh, David Orchard, and Daniel Veillard. Xml inclusions (xinclude) version 1.0. *W3C Working Draft*, 10, 2006.

[9] Markus Raab. A modular approach to configuration storage. *Master's thesis, Vienna University of Technology*, 2010.

[10] Markus Raab. Global and thread-local activation of contextual program execution environments. In *Proceedings of the IEEE 18th International Symposium on Real-Time Distributed Computing Workshops (ISORCW/SEUS)*, pages 34–41, April 2015.

[11] Markus Raab. Safe management of software configuration. In *Proceedings of the CAiSE'2015 Doctoral Consortium*, pages 74–82, urn:nbn:de:0074-1415-4, 2015. http://ceur-ws.org/Vol-1415/.

[12] Markus Raab and Franz Puntigam. Program execution environments as contextual values. In *Proceedings of 6th International Workshop on Context-Oriented Programming*, pages 8:1–8:6, NY, USA, 2014. ACM.

[13] Mark-Oliver Reiser. *Core Concepts of the Compositional Variability Management Framework (CVM): A Practitioner's Guide*. TU, Professoren der Fak. IV, 2009.

[14] Mark-Oliver Reiser. *Managing complex variability in automotive software product lines: subscoping and configuration links*. Südwestdt. Verlag für Hochschulschriften, 2009.

[15] Tavis Rudd, Mike Orr, and Ian Bicking. Cheetah-the python-powered template engine. In *10th International Python Conference.—2002*, 2007.

[16] J. Siméon and P. Wadler. The essence of XML. *ACM SIGPLAN Notices*, 38(1):1–13, 2003.

[17] Uwe Zdun. Tailorable language for behavioral composition and configuration of software components. *Computer Languages, Systems & Structures*, 32(1):56 – 82, 2006.

Efficiency considerations in heterogeneous cluster systems

Valon Raca and Eduard Mehofer

Faculty of Computer Science
University of Vienna, Austria
{valon.raca,eduard.mehofer}@univie.ac.at

Abstract. The importance of heterogeneous asymmetric clusters has grown steadily over the last years. Such architectures pose new challenges with respect to execution time and energy consumption. Work distribution for homogeneous systems has been done typically by dividing the work by the number of compute nodes and assigning equal-sized portions to each of them. Such an approach is not adequate any more for heterogeneous systems. The amount of work has to be adjusted to the computational power of the individual devices. Moreover, heterogeneity of devices raises in addition to execution time a second issue – the energy consumed to fulfill a task. A work distribution approach could e.g. exclude low-performing, high-energy-consuming devices from execution. Optimization in one direction only, i.e. execution time or energy consumption, does not meet the requirements. In this paper we discuss different work distribution approaches and report our experiences.

Keywords: heterogeneous clusters, runtime efficiency, energy efficiency

1 Introduction

The trend of using accelerators for solving time-consuming problems is continuing to increase steadily. Devices such as GPUs, Intel Xeon PHI, or FPGAs are performing much better than CPUs for many data-parallel classes of applications, thus becoming a mainstream architecture in HPC. However, heterogeneous systems require adequate support for handling programming obstacles resulting from the variety of hardware characteristics of those devices. The emergence of OpenCL has alleviated very much the burden of programmers dealing with different types of devices. OpenCL provides a platform for abstracting the hardware disparities through its transparent model which assists programmers writing codes which can be executed on any OpenCL enabled device.

Large clusters consisting of heterogeneous computing nodes with different computing devices provide the opportunity to scale-up the performance. OpenCL can adapt codes for a large number of devices from different vendors, but it does not support data transfers and coordination tasks within a cluster. In addition libraries like MPI are required. Further, the arrangement of accelerators in cluster compute nodes needs not to be symmetric. The devices can be distributed unevenly between the nodes resulting in an asymmetric configuration of a cluster.

Work distribution for such heterogeneous, asymmetric clusters becomes a major problem. Whereas for homogeneous clusters usually equal-sized portions of work has been distributed to the compute nodes, this approach is not adequate for heterogeneous systems any more. The amount of work has to be adjusted to the computational power of the individual devices. Moreover, energy-efficiency is another issue introduced by heterogeneous systems which is gaining increasing attention. A work distribution approach could e.g. exclude low-performing, high-energy-consuming devices from execution. Optimization in one direction only, i.e. execution time or energy consumption, does not meet the requirements. Energy-efficiency together with performance has to be taken into account in the time-energy problem space when dealing with efficiency issues.

We consider applications with a workload that is partitioned into work packages which are assigned to devices to be processed. Our programming and hardware model is described in detail in [12]. Runtime efficiency and energy efficiency basically boils down to the problem of mapping work packages to devices. Different work distributing strategies are possible resulting in different solutions exhibiting different efficiency characteristics. An assessment of a solution heavily depends on the requirements of the programmer. When we optimize in two directions, the mapping of work packages to devices results in a solution space containing 4 distinguished solutions exhibiting optimality criteria:

- *best performance*: one dimensional problem–energy consumption neglected
- *minimal energy*: one dimensional problem–execution time neglected
- *minimal energy with restrictions to performance*: two dimensional problem
- *best performance with restrictions to energy*: two dimensional problem

In addition to the 4 solutions above, trade-off solutions between time and energy can be considered as well. This papers discusses all the different solutions in detail and presents the impacts in practice.

The rest of the paper is organized as follows. Section 2 outlines our programming approach for heterogeneous, asymmetric clusters. Section 3 gives the motivation for efficiency considerations in heterogeneous asymmetric clusters. Optimization problems are discussed in Section 4. We survey the related work in Section 5 and conclude with a summary in Section 6.

2 Programming Support for Heterogeneous Asymmetric Clusters

In this section we give a brief overview of our framework [12] which supports heterogeneous, asymmetric cluster architectures. The goal of the framework is to assist a programmer to run an application efficiently in such an heterogeneous environment. Our strategy to deal with such clusters is to partition the work into work packages which are assigned to compute devices to be processed. As shown in Fig. 1, the input is partitioned in smaller data chunks called *work packages WP1,...,WPn* which are equal-sized. The size of work packages is determined by taking various hardware and application characteristics into account like memory capacity or peak performance.

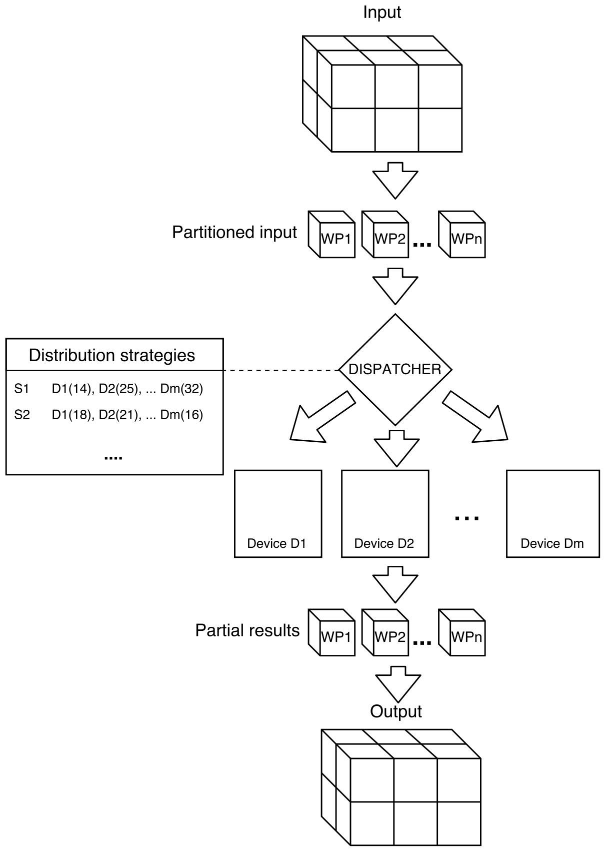


Fig. 1: Main application processing steps

Usually the number of work packages are orders of magnitude greater than the number of available devices in the system. This problem coupled with different performance and energy consumption numbers for each of the devices leads to the requirement for a strategy in distributing work packages. The different types of mappings is realized by a *dispatcher* which manages the distribution of work packages. As shown in Fig. 1, the dispatcher can perform different distribution strategies ($S1, S2, ...$) which define different mappings; e.g. strategy $S1$ requires that 14 WPs are assigned to device $D1$, 25 WPs to $D2$, and so forth, whereas strategy $S2$ requires that 18 WPs are assigned to device $D1$ and 21

WPs to $D2$. The output is constructed from the partial results of the processed work packages. The framework helps the programmer to run applications in such heterogeneous environments without dealing with hardware configurations or communication issues explicitly.

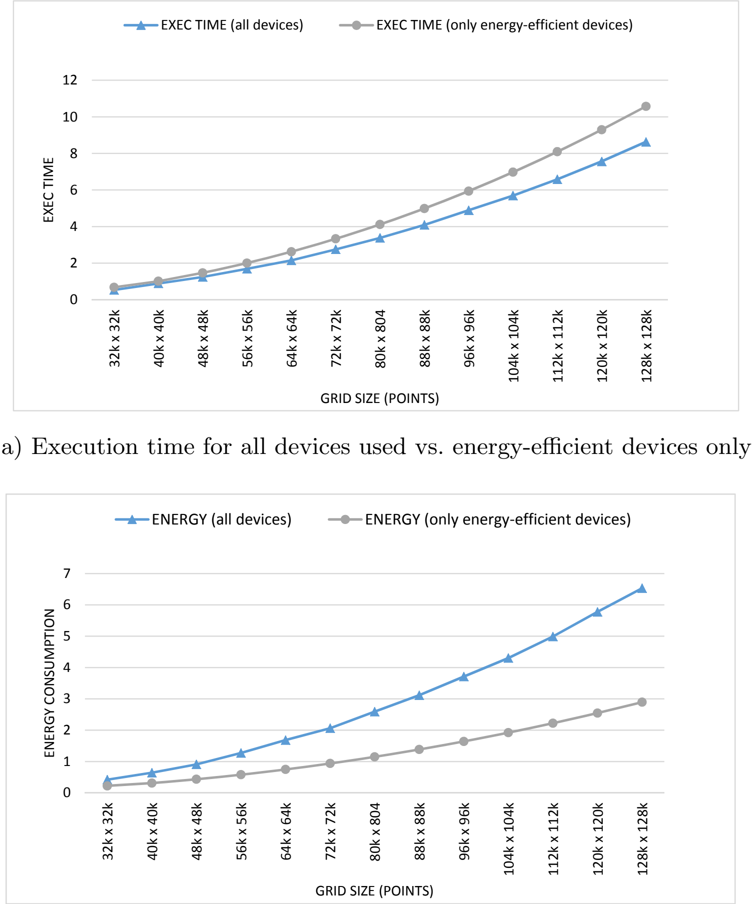
3 Examples for Selecting Devices and Distributing Work

As discussed above, work distribution is done by mapping work packages onto compute devices. Hence, different mappings result in different execution times and different power draws. Depending on whether execution time or energy consumption is favored by a programmer, some mappings fit better to the needs than others. In the following we will discuss three examples which have the common property that better performance is achieved at the cost of higher energy consumption. Or in other words: longer execution times may help to save energy. These examples show that getting better in one dimension may degrade the other dimension which means that an optimization has to take both dimensions into account.

The first example is taken from a paper by Liu and Luk [11]. As shown there, the FPGA is the most energy-efficient device for executing function SGEMM (matrix-matrix operation $C = \alpha AB + \beta C$) followed by GPU and CPU. Using only the FPGA for processing SGEMM leads to the smallest amount of energy consumption, but it may take about 40 times longer to complete than using the GPU.

We have done a similar experiment on our PHIA cluster which consists of 8 compute nodes with devices such as NVIDIA GPUs and Intel XEON Phi accelerators arranged in an asymmetric configuration. Our experimental results confirm that execution time and energy consumption can be influenced to a great extent based on the devices used in computing. In Fig. 2 we show the execution time and energy consumption in normalized units for a molecular analysis kernel with different problem sizes. As it is shown in Fig. 2a when we use only most energy-efficient devices the execution time (circular markers) increases in comparison to the program version where we use all available devices of the system (triangular markers). The gap between the two execution scenarios increases steadily with bigger problem sizes. However, in Fig. 2b it can be seen that energy consumption is reduced significantly when only the most energy-efficient devices are used in computation (circular markers). The figures show that for all problem sizes a loss in performance is rewarded by a reduction of energy consumption.

In addition to selecting devices for execution, the next example shows that the distribution of work onto the selected devices plays an important role as well. To demonstrate this, let us assume we have 5 equally-sized work packages and two devices with the following time and energy values per work package $T = \{2, 5\}$ and $E = \{10, 5\}$, i.e. 2 and 5 time units are taken for a work package on device 1 and 2, and 10 and 5 energy units on the devices, respectively. Further a time constraint is specified which requires that execution should finish within



(a) Execution time for all devices used vs. energy-efficient devices only

(b) Energy consumption for all devices used vs. energy-efficient devices only

Fig. 2: Performance and energy figures in our PHIA cluster for DCS

10 time units. The following mappings of work packages onto devices are feasible and fulfill the time constraint:

- If we assign 4 work packages to the first device and 1 work package to the second device, the overall execution time equals to $max\{8, 5\} = 8$ units of time, while the overall energy consumption equals to $10 * 4 + 5 = 45$ units of energy consumption.
- If we assign 3 work packages to the first device and 2 work packages to the second device, the overall execution time equals to $max\{6, 10\} = 10$ units of time, while the energy consumption equals $10 * 3 + 5 * 2 = 40$ units.

Both work distributions satisfy the time constraint, however the second work distribution is more energy-efficient than the first one.

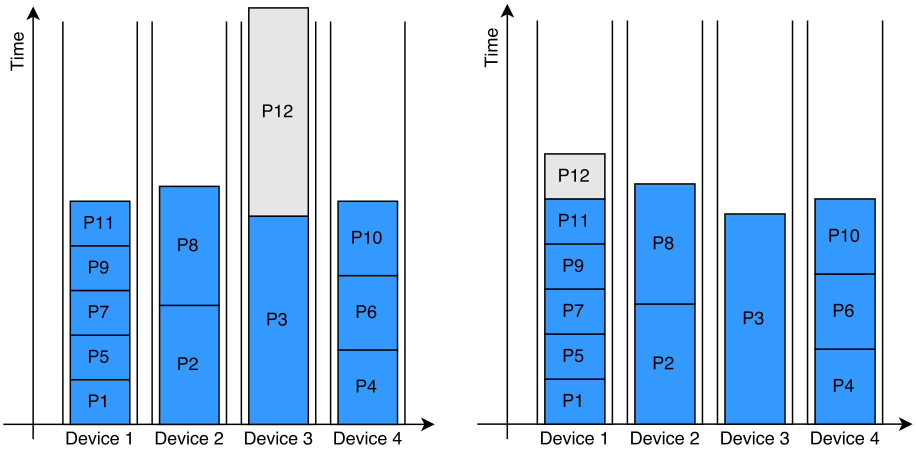
4 Different Solutions Satisfying Optimality Criteria

Best performance. In principle the best performance approach leads to a solution where all devices are used for computation. However, there are situations where this might lead to sub-optimal solutions. Consider a system as depicted in Fig. 3 with four devices with different processing times for work packages as indicated by the different lengths of the bars and 12 equal-sized work packages. In Fig. 3a the work packages are distributed at runtime to the devices just on request basis. Unfortunately, device 3 gets assigned the last work package 12, since the other devices are still busy.

However, a better distribution of work packages exists as shown in Fig. 3b. Although device 3 finished processing and is idle, it should not request another work package, but leave it to faster devices. Thus a simple on-request work package scheduler is not sufficient and more sophisticated techniques are required.

Minimal energy. An optimization directed at achieving minimal energy consumption tends to "serialize" the execution of an application. Minimal energy consumption means that only the most energy-efficient devices are used. As a consequence, in an heterogeneous environment only one type of devices with best energy-efficiency will be used. If only one instance of that device type exists in a cluster, the application will be executed on one device only.

Minimal energy with restrictions to performance. Since the minimal energy approach yields an extreme solution, there is a need for a relaxed approach. One possibility is to define a maximal execution time which shall be met with minimal energy preventing in this way the "serializing" behavior. As shown in Fig. 4a the programmer sets a time constraint for which the most energy-efficient distribution shall be found. All distributions below the dotted line meet the time constraint with the circular point being the most energy-efficient one.



(a) Simple request-based mapping

(b) More sophisticated mapping

Fig. 3: Different mappings of work packages onto devices

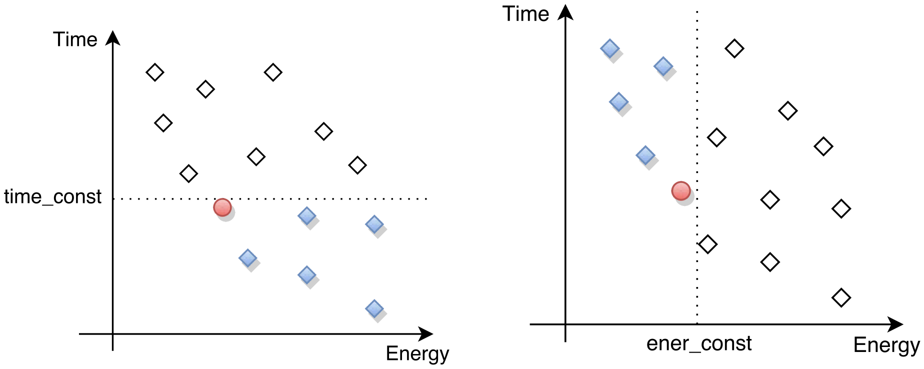
Best performance with restrictions to energy. Similar considerations as above lead to the definition of a maximal amount of energy which shall be met with best execution time possible. As shown in Fig. 4b the programmer sets an energy constraint for which the most time-efficient distribution shall be found. All distributions left to the dotted line meet the energy constraint with the circular point being the most time-efficient one.

Time-energy trade-off. Next we address the problem that no constraints have been specified by the programmer. Our goal is to propose a "good" solution by reasoning about time-energy trade-offs.

Fig. 5a shows the set of all possible distributions and Fig. 5b the corresponding Pareto front. It is reasonable to assume that a distribution of the Pareto front with minimal distance from the origin is a good compromise between execution time and energy consumption–the selected distribution is highlighted with a circular point.

5 Related work

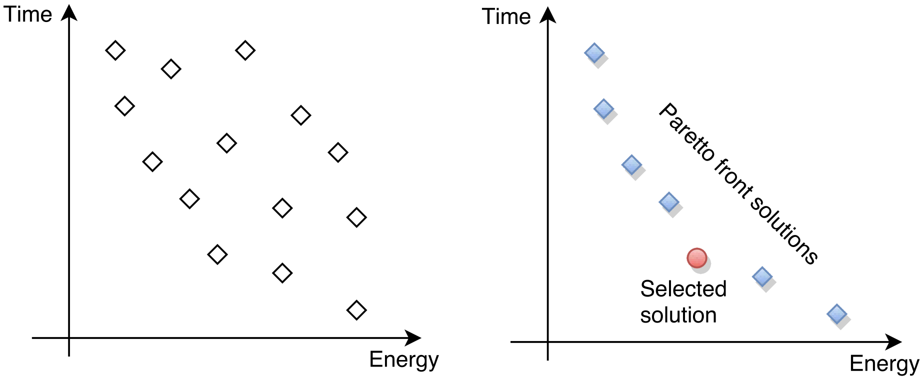
Many research efforts have been undertaken to support cluster systems, but most of them do not address optimizations taking both execution time and energy consumption into account in an heterogeneous, asymmetric hardware environment. One of the first cluster frameworks have been developed by Duato *et al.* [5] for CUDA and Barak *et al.* [3] for OpenCL. A more recent approach is SnuCL [8] which enables viewing of remote devices as part of a local context, while abstracting the communication between cluster nodes. libWater [6] is similar to



(a) Minimal energy with time constraint

(b) Best performance with energy constraint

Fig. 4: Solutions with constraints



(a) All possible distributions

(b) Pareto optimal distributions

Fig. 5: Set of possible solutions and Pareto front optimal solutions

the SnuCL approach, though it provides more efficient communication between cluster nodes, which is handled transparently by its runtime. Hybrid OpenCL [2] is based on FOXC runtime and enables communication between different OpenCL implementations in the context of distributed computing, while our framework targets HPC clusters. dOpenCL [7] allows using of different compute devices in any of the cluster nodes in a single application. It provides a central device manager which manages the assignment of the devices. clOpenCL [1] uses a wrapper library similar to dOpenCL targeting HPC clusters. DistCL [4] is similar to our approach, as it intends to abstract the multiple-devices in a cluster providing the programmer a single-device view for launching a kernel. However, DistCL runs its experiments across a symmetric cluster and only with one GPU per cluster. VOCL [13] provides a virtualization framework for GPU clusters using remote GPUs through proxy processes in cluster nodes. pVOCL [9] utilizes the VOCL framework and provides means for reducing energy consumption on a cluster. However, this approach is restricted to GPU clusters and it does not take into account actual energy consumption of applications but it is based on a table power model which needs to be provided by data center administrators. Whereas the cluster approaches presented so far have their origin in OpenCL, HeteroMPI [10] is an extension to MPI to support heterogeneous networks of computers. HeteroMPI provides the functionality to enable an application developer to deal with a heterogeneous environment and to realize a required behavior, but does not support the programmer with advanced features.

6 Conclusion

In this paper we discussed in detail the problem of distributing work onto devices of an heterogeneous, asymmetric cluster when both execution time and energy consumption should be taken into account. We presented different distribution strategies and discussed their implications in practice.

References

1. Alves, A., Rufino, J., Pina, A., Santos, L.: clOpenCL - Supporting Distributed Heterogeneous Computing in HPC Clusters. In: Euro-Par 2012: Parallel Processing Workshops, LNCS, vol. 7640, pp. 112–122. Springer Berlin Heidelberg (2013)
2. Aoki, R., Oikawa, S., Nakamura, T., Miki, S.: Hybrid OpenCL: Enhancing OpenCL for Distributed Processing. In: International Symposium on Parallel and Distributed Processing with Applications (ISPA). pp. 149–154 (May 2011)
3. Barak, A., Ben-Nun, T., Levy, E., Shiloh, A.: A package for OpenCL based heterogeneous computing on clusters with many GPU devices. In: 2010 IEEE Conference on Cluster Computing Workshops and Posters. pp. 1–7 (Sept 2010)
4. Diop, T., Gurfinkel, S., Anderson, J., Jerger, N.: DistCL: A Framework for the Distributed Execution of OpenCL Kernels. In: IEEE Symposium MASCOTS. pp. 556–566 (Aug 2013)
5. Duato, J., Pena, A., Silla, F., Mayo, R., Quintana-Orti, E.: rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In: HPCS. pp. 224–231 (June 2010)

6. Grasso, I., Pellegrini, S., Cosenza, B., Fahringer, T.: LibWater: Heterogeneous Distributed Computing Made Easy. In: ICS. pp. 161–172. Eugene, Oregon (2013)
7. Kegel, P., Steuwer, M., Gorlatch, S.: dOpenCL: Towards a Uniform Programming Approach for Distributed Heterogeneous Multi-/Many-Core Systems. In: Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW). pp. 174–186 (May 2012)
8. Kim, J., Seo, S., Lee, J., Nah, J., Jo, G., Lee, J.: SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters. In: Proceedings of the 26th ACM International Conference on Supercomputing (ICS). pp. 341–352. San Servolo Island, Venice, Italy (2012)
9. Lama, P., Li, Y., Aji, A.M., Balaji, P., Dinan, J., Xiao, S., Zhang, Y., Feng, W.c., Thakur, R., Zhou, X.: pVOCL: Power-Aware Dynamic Placement and Migration in Virtualized GPU Environments. In: Proceedings of ICDCS. pp. 145–154. Philadelphia, USA (2013)
10. Lastovetsky, A., Reddy, R.: HeteroMPI: Towards a message-passing library for heterogeneous networks of computers. Journal of Parallel and Distributed Computing 66(2), 197 – 220 (2006)
11. Liu, Q., Luk, W.: Heterogeneous systems for energy efficient scientific computing. In: Reconfigurable Computing: Architectures, Tools and Applications, LNCS, vol. 7199, pp. 64–75. Springer Berlin Heidelberg (2012)
12. Raca, V., Mehofer, E.: Device-Sensitive Framework for Handling Heterogeneous Asymmetric Clusters Efficiently. In: 26th IEEE International Symposium on Computer Architecture and High Performance Computing. Florianopolis, Brazil (Oct 2015)
13. Xiao, S., Balaji, P., Zhu, Q., Thakur, R., Coghlan, S., Lin, H., Wen, G., Hong, J., chun Feng, W.: VOCL: An optimized environment for transparent virtualization of graphics processing units. In: InPar. pp. 1–12 (May 2012)

An Optimizing Translation Framework for Strongly Mobile Java

Arvind Saini and Gerald Baumgartner

Division of Computer Science and Engineering
School of Electrical Engineering and Computer Science
Louisiana State University
Baton Rouge, LA 70803, USA

Abstract. Strongly mobile agents provide a convenient abstraction mechanism for migrating applications to the location of their data or as a container for deploying computational tasks in a cloud computing environment. They are difficult to implement on a stock Java VM, however, since it does not allow the computation state to be captured. We describe an implementation approach that translates strongly mobile Java into weakly mobile Java in which the generated code maintains serializable run-time stacks for the agent threads at all times. We discuss the optimizations that are needed for generating efficient weakly mobile code.

1 Introduction

For certain distributed applications, mobile agents (or mobile objects) provide a more convenient programming abstraction than remote method invocation (RMI). If an application needs to process large amounts of remote data, it may be less communication intensive to ship the computation in the form of a mobile agent to the location of the data than to use RMI calls to get the data and perform the computation locally. Mobile agents are also less affected by network connectivity. While the mobile agent is computing at a remote site, the home machine does not need to remain connected to the internet, which is especially useful if the home machine is a mobile device.

In mobile agent applications, agents typically operate autonomously using one or more threads that conceptually run within the agent. Existing mobile agent libraries for Java, such as Aglets [11, 10] or ProActive [2], however, only provide support for *weak mobility*, which allows migrating the agent object but requires that all threads are terminated before migration. However, *Strong Mobility*, which allows an agent to migrate seamlessly with running threads, would be the preferable programming abstraction. It allows a more natural programming style, since the logic for how and when an agent should migrate can be expressed procedurally and since it does not require the programmer to manually terminate all threads before migration and restart them at the destination. It also separates the migration mechanism from the application logic. Strong mobility, unfortunately, is difficult to implement since the Java Virtual Machine (VM) does not provide access to the run-time stacks of threads.

In previous research, we implemented support for strong mobility as a source-to-source translator from strongly mobile Java into weakly mobile Java [6, 20]. We also demonstrated that strongly mobile agents can be used as containers for deploying applications on a desktop grid [4, 5] or in the cloud [14]. They allow migrating an application that is encapsulated within the agent without the application programmer having to be aware of the migration.

Our mobility translator generates weakly mobile code by implementing the run-time stack of a thread as a serializable Java data structure. Compared to other approaches to strong mobility this has the advantage that it allows multi-threaded strongly mobile agents without modifying the Java VM. The disadvantage, however, is that it results in very inefficient code. Since a run-time stack is modified by the thread that owns it as well as by a thread that wants to migrate the agent, a locking mechanism is required to protect the integrity of the stacks. With fine-grained locking, this results in a high run-time overhead.

In this paper, we describe an optimization framework for our mobility translator. We present measurements for comparing the cost of different locking mechanisms. We also present a translation approach that can improve the performance of the generated code in exchange for a higher latency for migrations. Finally, we outline how standard compiler optimization techniques can be used for further optimizing the code.

In the next section, we provide more background on strong vs. weak mobility. Section 3 discusses related work on strong mobility. We explain our language and API design in Section 4 and the details of our mobility translator in Section 5. Section 6 presents experimental results on the potential speed improvements for mobile agents and Section 7 provides concluding remarks.

2 Background

Mobile agents and remote method invocation have the same expressive power. Any agent program can be translated into an equivalent RMI program and vice versa. In fact, either mechanism can be implemented on top of the other. Similar to loops and recursion, however, some problems are more naturally expressed in one of these programming styles.

In actual implementations, RMI is implemented on top of TCP together with object serialization to allow objects to be sent as arguments to remote methods. An agent migration is then implemented by the agent environment on the home machine performing a remote method invocation on the agent environment of the destination machine and passing the agent itself as argument to the remote method. In the case of weak mobility, only the agent object is sent to the destination. For strongly mobile agents, the execution state must be transferred as well.

A language with support for strong mobility provides an simple mental model for writing mobile agents. As an example, consider a network broadcast agent that prompts the user for input, relaying the input message to a number of other host machines. Using a Java-like language supporting strong mobility the solution is straightforward:

```
public void broadcast(String hosts[]) {
    System.out.println("Enter message:");
    String message = System.in.readln();
    for(int i = 0; i < hosts.length; i++) {
        try {
            dispatch(hosts[i]);
            System.out.println(message);
        }
        catch(Exception exc) {}
    }
    dispose();
}
```

Weak mobility does not allow migration of the execution state of methods (i.e., local variables and program counters). The dispatch operation simply does not return. Instead, the framework allows the developer to tie code to certain mobility-related events. E.g., in IBM's Aglets framework, the developer can provide callback code that will execute when an object is first created, just before an object is dispatched, just after an object arrives at a site, etc. Consider the above application written in an Aglets-like framework:

```
private String hosts[];
private int i = 0;
private String message;

public void onCreation(String hosts) {
    this.hosts = hosts;
    System.out.println("Enter message:");
    message = System.in.readln();
}

public void onArrival() {
    System.out.println(message);
}

public void run() {
    if(i == hosts.length)
        dispose();
    dispatch(hosts[i++]);
}
```

Because weak mobility does not allow the execution state to be transferred, programmers must manually store the execution state in agent fields (which are transferred) and must reconstruct the information of where the agent is and what it needs to do next using the event handling methods. This scatters the logic for how the agent moves from host to host across multiple methods and, therefore, results in an unnatural and difficult programming style.

While weak mobility is a conceptually simple mechanism and relatively straightforward to implement, it results in complex mobile agent code that may have to be written by expert programmers. By contrast, strong mobility provides a simple programming paradigm but it is more difficult to implement, e.g., to ensure freedom of race conditions and deadlocks.

3 Related Work

There are two main techniques for implementing strong mobility: modifying the Java VM or via translation of either source code or bytecode.

Java Threads [3] , D'Agents [8], Sumatra [1], Merpati [17], and Ara [13] extend the Sun JVM. CIA [9] modifies the Java Platform Debugger Architecture. JavaThread, CIA, and Sumatra to not support *forced migration*, i.e., the ability of an outside thread or agent dispatching an agent. Also, D'Agents, Sumatra, Ara, and CIA do not support the migration of multi-threaded agents. NOMADS [18] uses a customized virtual machine called Aroma that supports forced mobility and multi-threaded agent migration. The drawback of all these approaches is that relying on a modified or customized VM make it difficult to port and deploy agent applications. NOMADS and Java Threads are only compatible with JDK 1.2.2 and below, D'Agents needs the modified Java 1.0 VM, and Merpati and Sumatra are no longer supported. Furthermore, NOMADS, Sumatra, and Merpati do not support just-in-time compilation.

WASP [7] and JavaGo [16] implement strong mobility in a source-to-source translator that constructs a serializable stack just before the migration using the exception handling mechanism. Neither system is able to support forced mobility. Also, JavaGo does not support multi-threaded agent migration and does not preserve locks on migration. Correlate [19] and JavaGoX [15] are implemented using byte code translation. While they support forced mobility, they do not support multi-threaded agent migration.

Instead of using a source-to-source or bytecode translator for creating a serializable stack before migration like the previous translation approaches, in our approach a source-to-source translator ensures that serializable stacks are maintained at all times [6, 20]. This allows both forced migration and multi-threaded agent migration. Also, our approach better maintains the Java semantics, e.g., by preserving synchronization locks across migrations.

4 Language and API Design

Unlike a weak mobility library, which requires several event handlers and utility classes to simplify programming of itineraries, strong mobility can be supported with a very simple API. Our original support for strong mobility consisted simply of the interface `Mobile` and the two classes `MobileObject` and `ContextInfo`. While the design looks like a library API, it is really a language extension, since our proposed translation mechanism compiles away the interface `Mobile` and the class `MobileObject`.

4.1 Basic Mobility Support

Every mobile agent must (directly or indirectly) implement the interface `Mobile`. Similar to Java RMI, a client of an agent must access the agent through an interface variable of type `Mobile` or a subtype of `Mobile`.

Interface `Mobile` is defined as follows:

```
public interface Mobile extends java.io.Serializable {
    public void go(java.net.URL dest)
```

```
        throws java.io.IOException,
               com.ibm.aglet.RequestRefusedException;
}
```

Like Serializable, interface Mobile is a *marker interface*. It indicates to a compiler or preprocessor that special code might have to be generated for any class implementing this interface.

As explained in Section 5 below, we used the IBM Aglets library for implementing our support for strong mobility. This is currently reflected in the list of exceptions that can be thrown by go(). In a future version, we will add our own exception class(es) so that the surface language is independent of the implementation.

Class MobileObject implements interface Mobile and provides the two methods getContextInfo() and go(). To allow programmers to override these methods, they are implemented as wrappers around native implementations that are translated into weakly mobile versions.

```
public class MobileObject implements Mobile {
    private native ContextInfo realGetContextInfo();
    private native void realGo(java.net.URL dest)
        throws java.io.IOException,
               com.ibm.aglet.RequestRefusedException;
    protected ContextInfo getContextInfo() {
        return realGetContextInfo();
    }
    public void go(java.net.URL dest)
        throws java.io.IOException,
               com.ibm.aglet.RequestRefusedException {
        realGo(dest);
    }
}
```

A mobile agent class is defined by extending class MobileObject.

The method getContextInfo() provides any information about the context in which the agent is currently running, including the host URL and any system objects or resources that the host wants to make accessible to a mobile agent.

The method go() moves the agent to the destination with the URL dest. This method can be called either from a client of the agent or from within the agent itself. If go() is called from within an agent method foo(), the instruction following the call to go() is executed on the destination host. Typically, an agent would call getContextInfo() after a call to go() to get access to any system resources at the destination.

A mobile agent class could then simply be defined as a subclass of class MobileObject and would typically contain a thread that carries out the agent actions and moves to remote machines when needed.

4.2 Language Extensions

In a non-mobile applications, static fields of a class are shared between all the instances of that class. I.e., only one copy of a static field exists in the application. In a mobile application, when the code is migrated to a remote machine together with an agent, a new copy of a static field will be created at the destination. Depending on the use of the static field this may or may not be the desirable behavior. If a static field is used, say, to count the number of instances of a class, it may be preferable to have a globally unique field. Similar to static fields, we propose a declaration for global variables,

```
global int n;
```

such that an agent always accesses global variable on the home machine, not on the machine the agent currently resides on.

We also propose a new language construct `immobile`, both as a modifier for methods and as a block of code,

```
immobile int foo() { ... }
int bar() { ... \immobile{ ... } ... }
```

that inhibits migration of an agent while it is executing immobile code. This gives programmers better control of mobility for multi-threaded agents, e.g., by postponing a migration until a large intermediate data structure has been deallocated.

4.3 Mobile Threads and Thread Pools

In our original implementation of strong mobility for multi-threaded agents, we used `Thread` and `ThreadGroup` objects in the strongly mobile code and generated the wrapper classes `MobileThread` and `MobileThreadGroup` as part of the weakly mobile code. These mobile threads were not available to programmers, though. We are exploring a redesign of the API that would use either thread pools or executors instead of thread groups and that would make a mobile thread API available to the programmer. This would allow programmers to use serializable mobile threads standalone without using agents.

Migration for mobile agents is a similar problem to checkpointing a high performance computing application. With checkpointing, a thread is serialized and written to disk. If the processor fails, the thread is read back in and restarted. This is the same mechanism needed for thread migration with the disk acting as a very slow communication link. Providing an API for mobile threads and thread pools would allow our mobility translator to be used for automatically generating checkpointing code.

5 Translation from Strong to Weak Mobility

5.1 Single-Threaded Agents

For efficiency reasons it would be desirable to provide virtual machine support for strong mobility. However, a preprocessor or compiler implementation has the advantage that the generated code can run on any Java VM, and that it is easier to implement and to experiment with the language design.

For our initial prototype, we chose to design the translation mechanism for a preprocessor that translates strongly mobile code into weakly mobile code that uses the

Aglets library. For our current reimplementation, we will generate code for the ProActive library [2].

For implementing strong mobility in a preprocessor, it is necessary to save the state of a computation before moving an agent so it can be recovered afterwards. Fünfrocken describes a translation mechanism that inserts code for saving local variables just before moving the agent [7]. This has the disadvantage that the `go()` method cannot be called from arbitrary points outside the agent.

Our translation approach is to maintain a serializable version of the computation state at all times by letting the agent implement its own run-time stack. This increases the cost of regular computation as compared to Fünfrocken's approach, but it simplifies restarting the agent at the remote site.

5.2 Translation of Methods

For making the local state of a method serializable, we implement activation records of agent methods as objects. For each agent method, the preprocessor generates a class whose instances represent the activation records for this method.

An activation record class for a method is a subclass of the abstract class `Frame`:

```
public abstract class Frame
    implements Cloneable, java.io.Serializable {
    public Object clone() { ... }
    abstract void run();
}
```

Activation records must be cloneable for implementing recursion as explained below. The translated method code will be generated in method `run()`.

For example, given an agent class `C` with a method `foo` of the form

```
void foo(int x) throws AgletsException {
    int y = x + 1;
    go(new URL(dest));
    System.out.println(y);
}
```

(and ignoring exception handling and synchronization for simplicity) we might generate a class `Foo` of activation records for `foo` of the form

```
class Foo extends Frame {
    C This;
    int x;
    int y;
    int pc = 0;          // program counter

    Foo(C t) { this.This = t; }
    void setArgs(int x) { this.x = x; }
    void run() {
        if (pc == 0) { pc++;  y = x + 1; }
        if (pc == 1) { pc++;
```

```
                       go(new URL(This.dest)); This.run1(); }
        if (pc == 2) { pc++;  System.out.println(y); }
    }
}
```

The parameter and the local variable of method `foo()` became fields of class `Foo`. In addition, we introduced a program counter field `pc` and a variable `This` for accessing fields in the agent object.

The method `run()` contains the original code of `foo()` together with code for incrementing the program counter and for allowing `run()` to resume computation after moving. Calls of agent methods are broken up into a call of the generated method followed by `This.run1()`, as explained below. For allowing the agent to be dispatched by code outside the agent class, the program counter increment and the following instruction must be performed atomically, which requires additional synchronization code.

For efficiency, the preprocessor could group multiple statements into a single statement and only allow the agent to be moved at certain strategic locations.

5.3 Translation of Agent Classes

An agent now must carry along its own run-time stack and method dispatch table. The generated agent class contains a `Frame` array as a method table and a `Stack` of `Frame`s as the run-time stack. When calling a method, the appropriate entry from the method table is cloned and put on the stack. After passing the arguments, the `run` method executes the body of the original method `foo` while updating the program counter.

Suppose we have an agent class `AgentImpl` of the form

```
public class AgentImpl extends MobileObject implements Agent{
    int a;
    public AgentImpl() { /* initialization code */ }
    public void foo(int x) throws AgletsException { ... }
}
```

Since this class indirectly implements interface `Mobile`, the preprocessor translates it into the following code:

```
public class AgentImpl extends Aglet {
    int a;
    Frame[] vtable = { new Foo(this) };
    final int _foo = 0;
    Stack stack = new Stack();

    public void onCreation (Object init) {
        /* initialization code */
    }

    public void foo(int x) {
        Foo frame = (Foo) (vtable[_foo].clone());
        stack.push(frame);
        frame.setArgs(x);
    }

    public void run1() {
        Frame frame = (Frame) stack.peek();
        frame.run();
        stack.pop();
    }

    class Foo extends Frame { /* as described above */ }
}
```

The preprocessor eliminates interface `Mobile` and class `MobileObject` and lets the agent class extend class `Aglets`.

For implementing method dispatch, the agent includes a method table `vtable` of type `Frame[]`. The constant `_foo` is the index into the method table for method `foo`. The field `stack` implements the run-time stack.

The constructor of class `AgentImpl` is translated into the method `onCreation`. Since Aglets only allows a single `Object` as argument of `onCreation()`, any original constructor arguments must be packaged in an array or vector by the preprocessor.

As described above, the original agent method `foo()` gets translated into a local class `Foo` of activation records. The method `foo()` in the generated code implements the call sequence: it allocates an activation record on the stack and passes the arguments. The code for executing the method on the top of the stack and for popping the activation record in method `run1()` is shared between all methods. A client must first call `foo()` followed by a call to `run1()`.

For resuming execution after arriving at the destination, we must also generate a method `run()` inside class `AgentImpl`:

```
public void run() {
    while (! stack.empty())
        run1();
}
```

5.4 Protection of Agent Stacks

It is imperative that an agent cannot be dispatched by another thread between incrementing the program counter and executing the following statement. If the program counter increment and the following statement were not executed atomically, a thread could be dispatched after the program counter increment and incorrectly miss execution of the statement upon arrival. Since by definition this type of synchronization need not be maintained across VM boundaries, standard Java synchronization techniques are used. For a single-threaded agent, we simply synchronize on the agent object itself. For method calls, we only need to protect the call to set up the activation record. The actual execution of `run1()` does not need to be synchronized since by then a new activation record with its own `pc` will be on top of the stack:

```
synchronized(This) { pc++; go(new URL(This.dest)); }
This.run1();
```

For preventing the agent from being dispatched between the program counter increment and the next instruction, the call of `realGo()` in `MobileObject.go()` must also be synchronized on the agent object.

If two agents try to dispatch one another, this synchronization code could lead to a deadlock. For executing the statement `b.go(dest)`, Agent `a` would first synchronize on itself. Then a synchronization on `b` would be required to protect the integrity of `b`'s stack. If similarly `b` would execute `a.go(dest)`, a deadlock would result. To prevent this, the call of `realGo()` is synchronized on the agent context instead of on the caller.

```
public class MobileObject implements Mobile {
    public void go(java.net.URL dest)
        throws IOException, RequestRefusedException {
        synchronized(TheAgentContext) {
            synchronized(this) { realGo(dest); }}
    }
```

The only time any thread synchronizes on two objects is now in the call of `realGo()`, in which case the first synchronization is on the agent context. Deadlocks are, therefore, prevented.

This synchronization mechanism ensures that only one agent can migrate at a time. If two agents `a` and `b` try to dispatch one another, the first one, say `a`, will succeed. By the time `b` tries to dispatch `a`, `a` is already on a different host. The call to `a.go()` will, therefore, throw an exception that must be handled by `b`.

5.5 Multi-Threaded Agents

Our mobility translator supports migration of multithreaded agents. Unfortunately, the Java library classes `Thread` and `ThreadGroup` are not serializable. Therefore, for each use of the classes `Thread` and `ThreadGroup` we need to generate a serializable wrapper of classes `MobileThread` and `MobileThreadGroup`, respectively. The `go()` method on an agent can be invoked by another agent in the system or by a thread within the agent itself. The `go()` method calls the `realgo()` method to check whether the agent is already on the move. If so, a `MoveInterrupt` exception is thrown. Otherwise, each `MobileThread` calls the `interrupt()` method of the underlying`Thread` class. This terminates any `wait()`, `join()`, or `move()` functions if they are being executed. The time remaining to completely execute these function calls is saved so that the function can resume execution at the destination from the point where it had been interrupted.

The next step is to call the `packUp()` method of the `main` agent wrapper of the thread group. This in turn calls the `packUp()` methods of the wrappers for all the threads and the thread groups. The underlying state of execution of each thread and thread group is saved to the corresponding wrappers. All the threads are forced to halt any further executions and subsequently the agent is shipped to the destination by the `dispatch()` call. At the destination, the `reinit()` method of the `main` agent thread group wrapper is invoked. This method calls the `reinit()` method of each wrapper. The called `reinit()` methods create `Thread` or `ThreadGroup` objects from their corresponding wrappers and the execution states of the threads are restored.

After the restoration of the execution states, the `start()` method of the `main` thread group wrapper is called. This method invokes the `start()` methods of all the `MobileThread` wrappers. Then `start()` method of the underlying thread is called, which then calls the `run()` method of the `MobileThread` wrapper. The `run()` method checks the stack of the `MobileThread` wrapper. If the stack is empty, then the `run()` method of the `Runnable` target is called. Otherwise, the activation records in the stack are executed.

5.6 Synchronization in Multi-Threaded Agents

An agent should not be shipped to the destination while a thread is in the middle of executing a statement. To prevent this from happening, the program counter update and a statement execution should be performed atomically. Neither should any two agents dispatch each other at the same time nor should two threads within the same agent try to move the agent simultaneously. For example, each statement in the thread is protected by a lock mechanism as shown below:

```
Acquire lock;
Program counter update;
Statement execution;
Release lock;
```

The problem of lock synchronization for multi-threaded agents is comparable to the readers-write problem with writers priority. Each thread in the agent is assigned a lock. The threads that are executing statements are considered to be readers and the thread that invokes the `go()` method to move the agent is considered to be the writer. After the reader thread is done executing the statement, the lock is released and acquired by the writer thread. When the writer thread has acquired the locks of all the readers, only then can the agent be allowed to relocate.

The drawback by having a locking mechanism around each program counter update and statement, is that it incurs a large overhead. On the other hand, synchronizing on an entire agent instance reduces the degree of parallelism in the system.

5.7 Optimizations

Our translation mechanism introduces several sources of inefficiencies. Migration of a strongly mobile agent is slower than that of a handwritten weakly mobile agent, because the run-time stacks need to be serialized and shipped along with the agent. However, since the expected behavior of mobile agents is that they spend a significantly larger amount of time computing than migrating, the overhead imposed on regular computation is of much more concern. The computation overhead comes from three sources: the locking mechanism for protecting the run-time stacks, the frequency of locking and the associated overhead of testing and incrementing the program counter, and pushing activation records onto the run-time stacks.

A straight-forward optimization is to combine multiple consecutive statements, e.g., multiple assignments, into a single block without releasing and re-acquiring the lock after each statement. This increases the latency slightly until a call to `go()` is honored and the agent can migrate, but given the infrequency and cost of migration, even a latency of up to 1 second would likely not be a problem for most applications.

Much of the locking overhead itself comes from insuring that writers (i.e., threads that want to move an agent) do not starve. A readers-writer lock with reader priority would be significantly cheaper but it could not insure freedom of starvation for writers. Since writers occur very infrequently, it is possible to keep the stack locked for readers by default and only allow a writer to proceed if one is pending. E.g., instead of releasing and re-acquiring the lock, we could use

```
if (Writer is present) { Release lock;  Acquire lock; }
```

using an atomic Boolean or atomic integer to test for the presence of writers.

Such a locking-scheme then allows a different code structure. Instead of having lock-unlock pairs around statements or consecutive groups of statements, it would be possible to have these if-conditions with unlock-lock pairs only in a few strategic places in the code. Again, this would increase the latency until a migration can take place, but it has the potential to drastically improve performance.

In addition, it would be possible to use standard compiler optimizations to further reduce the run-time overhead. The overhead of maintaining the program counter for a loop can be reduced by unrolling the loop. Inlining of methods can be used to eliminate the expensive method call sequence. Methods that do not contain loops may not need to be translated at all. Finally, with worst-case execution time analysis, it would be possible to give a bound on the run-time of a method or code fragment and only generate locking code to test for the presence of writers if the worst-case execution time is more than the acceptable migration latency.

6 Experiments

To indicate the overhead of our translation mechanism and the potential for optimizations, we first present the results of manual optimizations and measurements that had been performed in prior work [6]. These measurements were made on a quad-core UltraSparc-II 296MHz processor with 1GB of memory running Solaris and using the Sun JDK 1.4.0 Hotspot VM.

For these measurements, standard Java benchmarks were rewritten in the form of both strongly mobile agents and Aglets. This did not involve changing the timed code significantly. The only changes that needed to be made to the original benchmarking code were made to avoid method calls inside expressions, since the preprocessor did not yet handle these.

The strongly mobile agents were passed through the translator. We then used simple manual optimization techniques to improve the performance of the translated agents. These are: the grouping of simple statements to form logical, atomic statements; the acquiring and releasing of locks only every 10,000 simple statements for a loop; and the inlining of calls to simple methods that in turn do not contain method calls.

The running times and memory footprints of the translated agents and the manually optimized agents were compared with the equivalent weakly mobile Aglets. The results have been presented in Table 1. A major contributor to the poor running times of the recursive benchmark programs is the garbage collector that runs several times a second during their execution.

Table 1. Execution time of strongly mobile agents compared to corresponding Aglets code.

Benchmark	Translated Code	Optimized Code
Crypt (array size: 3,000,000, no threads)	5.61X	1.23X
Crypt (array size: 3,000,000, 1 thread)	5.96X	1.30X
Crypt(array size: 3,000,000, 2 threads)	6.00X	1.41X
Crypt(array size, 3,000,000, 5 threads)	5.60X	1.31X
Linpack (500 X 500)	10.00X	1.75X
Linpack (1000 X 1000)	9.48X	1.65X
Tak (100 passes)	245.30X	220.83X
Tak (10 passes)	247.00X	213.60X
Simple recursion (sum 1–100, 10,000 passes)	68.27X	60.75X

We performed further optimzations on the Linpack benchmark, a matrix multiplication implementation. The inner-most loop of Linpack is inside a dot-product method. We manually inlined this method, and measured execution time with the inner-most loop untranslated, and with the translated loop unrolled. The running time comparisons are presented in Table 2.

Table 2. Potential performance improvements for inner loop transformations of strongly mobile Linpack code relative to Aglets.

Linpack Version	Untranslated	Unrolled 2X	Unrolled 10X
Linpack (500 X 500)	1.02X	1.21X	0.75X
Linpack (1,000 X 1,000)	1.02X	1.15X	0.76X

For finding the cheapest locking mechanisms, we performed micro-measurements of lock-unlock pairs for several different locking mechanisms as well as using atomic integers or Booleans as guards for a lock. These measurements were performed on a quad-core, 2.4GHz Xeon workstation running Linux. Since all code is sequential and to make the measurements more predictable, we disabled multi-core support, hyper-threading, Intel Turbo Boost (overclocking), and Intel Speed Step (CPU throttling), and turned off all network interfaces, the X window system, and unnecessary background processes. The Java Version 1.7.0_21 and ran the measurements on the Java server VM with the command line options `-XX:CICompilerCount=1` and `-Xbatch` to ensure that the measurements are not distorted by background compilation. We took 100 measurements of 10,000 lock-unlock pairs each in a 10X-unrolled loop. The average times are shown in Table 3.

Table 3. Average execution time for one lock-unlock pair.

Locking Mechanism	Time (ns)
Semaphore	18.32
ReentrantLock	16.41
ReentrantReadWriteLock (Read Lock)	26.77
ReentrantReadWriteLock (Write Lock)	23.84
AtomicBool (as guard for lock)	16.09
AtomicInt (as guard for lock)	15.92

As our measurements show, the cheapest combination would be to use an atomic integer or Boolean (the difference between them is not statistically significant) as a guard for a `ReentrantLock` instead of our original counting `Semaphore`. With guarded locks it would be possible to generate code that unlocks and re-acquire the lock less frequently. This, together with compiler optimizations such as not translating inner loops or methods without loops, inlining, and loop unrolling has the potential to reduce the overhead to less than 20% for non-recursive applications, which would be acceptable.

7 Conclusion

We have presented a framework for translating strongly mobile Java code into weakly mobile code. Compared to existing approaches to strong mobility, our approach has the advantages that it allows multi-treaded agents and forced mobility, accurately maintains the Java semantics, and can run on a stock Java VM. The disadvantage is that without further optimizations, the run-time overhead would be prohibitively large.

The main contribution of this paper is that it presents an optimization framework for improving the performance of the generated weakly mobile code. Preliminary measurements show that with a combination of a cheaper locking mechanism and a code structure that trades off migration latency for performance, the overhead can become acceptably small. Finally, standard compiler optimization techniques can be used to further improve the performance of the generated code. We are currently working on a reimplementation of our mobility translator in the Polyglot compiler framework [12].

References

1. A. Acharya, M. Ranganathan, and J. Saltz. Sumatra: A language for resource-aware mobile programs. In J. Vitek, editor, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 111–130. Springer-Verlag, 1996.
2. F. Baude, D. Caromel, F. Huet, and J. Vayssìere. Communicating mobile active objects in Java. In M. Bubak, H. Afsarmanesh, R. Williams, and B. Hertzberger, editors, *Proceedings of HPCN Europe 2000*, volume 1823 of *Lecture Notes in Computer Science*, pages 633–643. Springer Verlag, May 2000.
3. S. Bouchenak, D. Hagimont, S. Krakowiak, N. D. Palma, and F. Boyer. Experiences implementing efficient Java thread serialization, mobility and persistence. In *Software — Practice and Experience*, pages 355–394, 2002.

4. A. J. Chakravarti and G. Baumgartner. Self-organizing scheduling on the Organic Grid. *Int. Journal on High Performance Computing Applications*, 20(1):115–130, 2006.
5. A. J. Chakravarti, G. Baumgartner, and M. Lauria. The Organic Grid: Self-organizing computation on a peer-to-peer network. *Trans. Sys. Man Cyber. Part A*, 35(3):373–384, May 2005.
6. A. J. Chakravarti, X. Wang, J. O. Hallstrom, and G. Baumgartner. Implementation of strong mobility for multi-threaded agents in Java. In *Proceedings of the International Conference on Parallel Processing*, pages 321–330. IEEE Computer Society, Oct. 2003.
7. S. Fünfrocken. Transparent migration of Java-based mobile agents: Capturing and reestablishing the state of Java programs. In K. Rothermel and F. Hohl, editors, *Proceedings of the Second International Workshop on Mobile Agents*, volume 1477 of *Lecture Notes in Computer Science*, pages 26–37, Stuttgart, Germany, September 1998. Springer-Verlag.
8. R. S. Gray, D. Kotz, G. Cybenko, and D. Rus. D'Agents: Security in a multiple-language, mobile-agent system. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 154–187. Springer-Verlag, 1998.
9. T. Illmann, T. Krueger, F. Kargl, and M. Weber. Transparent migration of mobile agents using the Java platform debugger architecture. In *Proceedings of the 5th International Conference on Mobile Agents*, MA '01, pages 198–212, London, UK, 2002. Springer-Verlag.
10. D. B. Lange and M. Oshima. Mobile agents with Java: the Aglets API. *World Wide Web Journal*, 1998.
11. D. B. Lange and M. Oshima. *Programming & Deploying Mobile Agents with Java Aglets*. Addison-Wesley, 1998.
12. N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In G. Hedin, editor, *Compiler Construction, 12th International Conference, CC 2003*, volume 2622 of *Lecture Notes in Computer Science*, pages 138–152. Springer-Verlag, Apr. 2003.
13. H. Peine and T. Stolpmann. The architecture of the Ara platform for mobile agents. In R. Popescu-Zeletin and K. Rothermel, editors, *First International Workshop on Mobile Agents*, volume 1219 of *Lecture Notes in Computer Science*, pages 50–61, Berlin, Germany, Apr. 1997. Springer Verlag.
14. B. Peterson, G. Baumgartner, and Q. Wang. A hybrid cloud framework for scientific computing. In *8th IEEE International Conference on Cloud Computing, CLOUD 2015*, pages 373–380, New York, NY, June 2015.
15. T. Sakamoto, T. Sekiguchi, and A. Yonezawa. Bytecode transformation for portable thread migration in Java. In *Proceedings of Agent Systems, Mobile Agents, and Applications*, volume 1882 of *Springer Verlag Lecture Notes in Comuter Science*, 2000.
16. T. Sekiguchi, H. Masuhara, and A. Yonezawa. A simple extension of java language for controllable transparent migration and its portable implementation. In *In Proceedings of the 3 rd Intl. Conference on Coordination Models and Languages*, 1999.
17. T. Suezawa. Persistent execution state of a Java virtual machine. In *Java Grande*, pages 160–167, 2000.
18. N. Suri, J. M. Bradshaw, M. R. Breedy, P. T. Groth, G. A. Hill, R. Jeffers, and T. S. Mitrovich. An overview of the NOMADS mobile agent system. In C. Bryce, editor, *6th ECOOP Workshop on Mobile Object Systems*, Sophia Antipolis, France, 13 June 2000.
19. E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen, and P. Verbaeten. Portable support for transparent thread migration in Java. In *Proceedings of the Joint Symposium on Agent Systems and Applications / Mobile Agents*, pages 29–43, Zurich, Switzerland, September 2000. Springer-Verlag.
20. X. Wang, J. Hallstrom, and G. Baumgartner. Reliability through strong mobility. In *Proc. of the 7th ECOOP Workshop on Mobile Object Systems: Development of Robust and High Confidence Agent Applications (MOS '01)*, pages 1–13, Budapest, Hungary, June 2001.

Entwurf und Implementierung einer Sprache zur Planung optimaler Pumpensysteme

Benjamin Saul, Wolf Zimmermann

Institut für Informatik
Martin-Luther-Universität Halle-Wittenberg
Von-Seckendorff-Platz 1
06120 Halle (Saale)

Zusammenfassung. In diesem Artikel wird eine Sprache und deren Umsetzung für die Planung optimaler Pumpensysteme zur Förderung von Flüssigkeiten vorgestellt. Die entwickelte domänenspezifische Sprache beschreibt Komponenten wie z. B. Pumpen und die zugehörigen Anforderungen. Wir beschreiben wichtige Sprachkonzepte und gehen insbesondere auf die Generierung eines linearen Problems zur Optimierung ein. Die Optimierungsaufgabe eines solchen Systems liegt darin, die bestmögliche Auswahl an zu installierenden Pumpen zu treffen, um möglichst energie- oder kosteneffizient die Anforderungen zu erfüllen. Vor und während der Generierung dieser Optimierungsaufgabe müssen Analysen ähnlich zu Programmanalysen erfolgen, um möglichst detaillierte Fehler- und Diagnosemeldungen zu generieren.

1 Motivation

In vielen industriellen Fertigungs- und Produktionsanlagen sowie Versorgungsnetzwerken kommen Pumpensysteme zum Einsatz. Ein Pumpensystem ist eine Anlage mit Pumpen, Rohren und diversen Armaturen, deren Zweck es ist, Flüssigkeiten mit Anforderungen an Druck und Durchfluss über eine Distanz zu transportieren. Eine Beispielanlange ist in Abbildung 1 dargestellt. Es gibt fest vorgegebene Komponenten, wie Behälter und Abnahmestellen, mit spezifischen Drücken und Durchflüssen, die entweder durch die Umgebung vorliegen oder erst erreicht werden müssen. Das Erreichen dieser Drücke geschieht durch den Einbau und Betrieb diverser Pumpen.

Das Betreiben von Pumpen stellt zusammen mit sonstigen Fördermethoden wie Fließbändern 37% der in Deutschland benötigten Energie dar [12]. Daher sollen die Anlagen möglichst so geplant werden, dass der Stromverbrauch und die Anlagenkosten möglichst gering sind. Optimierungen von industriellen Anlagen können Einsparpotentiale von bis zu 90% der Pumpenenergiekosten erzielt werden [4]. Um dieses Potential voll auszunutzen, wurde in [11] ein Verfahren entwickelt, mit dem man optimale Pumpensysteme berechnen lassen kann. Dabei muss ein abstraktes Pumpensystem, also sämtliche Anordnungen und Konfigurationen von Pumpen und Armaturen, als mathematischen Optimierungsproblem

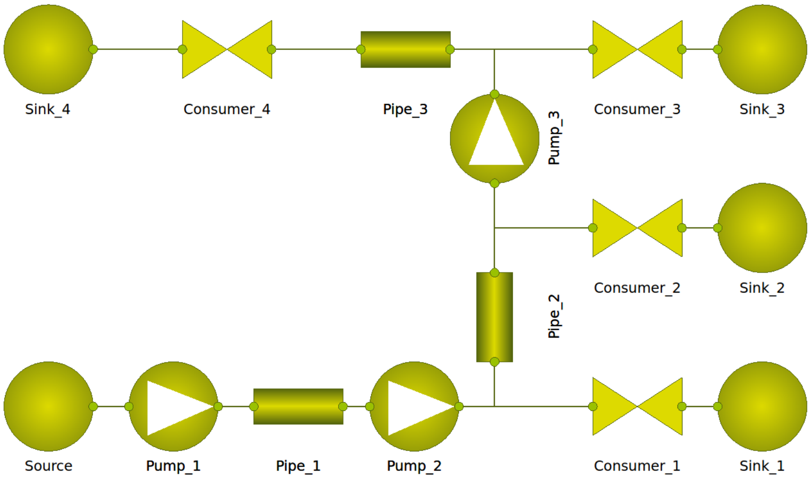


Abb. 1. Beispiel eines Pumpensystemes mit verschiedenen Pumpen und Abnehmern. Aufgabe ist es, einen Strom von der Quelle zu den Senken zu erzeugen und dabei bestimmte Drücke bei den Abnehmern zu erzielen.

formuliert werden. Dieses besteht zu einem großen Teil aus verschiedenen Variablen und Gleichungen, ist also für einen Pumpenanlagenplaner nur schwer handhabbar. Diese müssen dort Variablen für einzelne Pumpen definieren und aus dem Ergebnis der Optimierung herauslesen, wie die optimale Anlage beschaffen sein muss.

Um den Architekt der Pumpenanlage zu unterstützen und das Entwerfen einer optimalen Anlage zu einem durchgängigen Prozess zusammenzufassen, entwickeln wir die domänenspezifische Sprache SHEP (Sprache für hocheffiziente Pumpensysteme). Den zugehörigen Übersetzer entwickeln wir mit der Werkzeugsammlung Eli [8]. In der Sprache können Komponenten des Pumpensystems, wie Rohre, Pumpen und Ventile, als Typen beschrieben werden. Die darin geltenden physikalischen Gesetzmäßigkeiten sind bereits vordefiniert. Für das System können nun einzubauende Komponenten bestimmt werden. Mit Typprüfungen wird dann unter anderem bestimmt, welche Komponenten des Pumpensystems miteinander verschaltet werden können. Die aktuelle Hauptaufgabe des Übersetzers ist es, ein Optimierungsmodell aus der domänenspezifischen Beschreibung zu erstellen. Dieses wählt dann unter den möglichen Pumpen diejenigen aus, welche im optimalen System eingesetzt werden sollen. Die benötigten Aspekte des Übersetzers werden in Abbildung 2 dargestellt.

Erzeugen von linearen Problemen aus domänenspezifischen Sprachen Hierbei kommt es im Wesentlichen darauf an, dass Verhaltensgleichungen möglichst günstig beschrieben werden, damit die Optimierung schnell durchgeführt werden kann. Die Attribute der einzelnen Komponenten müssen als Variable und Parameter des Optimierungsproblems formuliert werden. Ent-

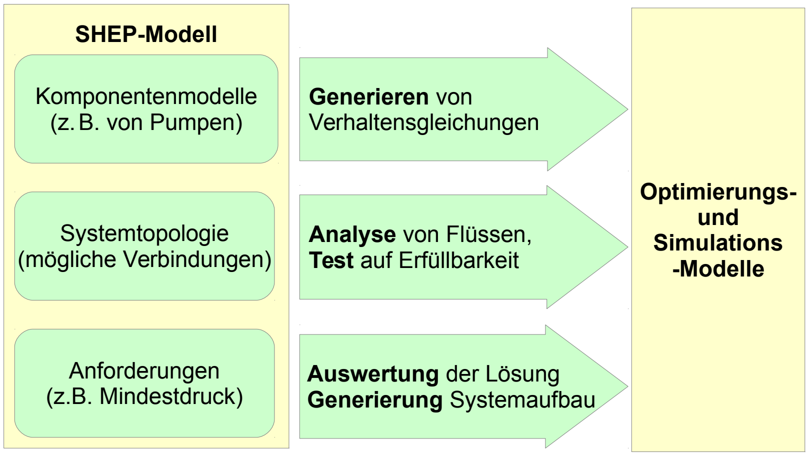

SHEP-Modell
Komponentenmodelle (z. B. von Pumpen)
Systemtopologie (mögliche Verbindungen)
Anforderungen (z.B. Mindestdruck)
Generieren von Verhaltensgleichungen
Analyse von Flüssen, Test auf Erfüllbarkeit
Auswertung der Lösung Generierung Systemaufbau
Optimierungs- und Simulations -Modelle

Abb. 2. Übersicht über die Werkzeugkette zur Erstellung optimaler Pumpensysteme.

sprechende Gleichungen für z. B. Flusserhaltung müssen generiert und linearisiert werden.

Analyse des Flüssigkeitsstromes und Lösbarkeitstest Durch Vorabanalysen im System werden unzulässige Verbindungen detektiert, welche die daran anliegenden Drücke oder Durchflüsse nicht unterstützen. Der Test auf Lösbarkeit ist notwendig, denn die Fehlermeldungen eines Lösers sind für den Systemarchitekten unzureichend. Die Fehlermeldungen geben nur die Unlösbarkeit an, ohne deutlich machen zu können, welche Druckanforderung den Widerspruch ausgelöst hat. Das Wissen der Domäne kann hier detaillierte Fehlermeldungen möglich machen.

Generieren des optimierten Pumpensystems aus der Lösung Ergebnis des Lösers sind die Belegungen der Variablen, insbesondere der Entscheidungsvariablen. Diese zu interpretieren und auszuwerten ist manuell schwierig und fehleranfällig. Daher müssen diese im Übersetzer gesammelt werden und daraus das endgültige optimierte geplante Pumpensystem formuliert werden. Dieses wird als Simulationsmodell beschrieben, um überprüfende Berechnungen darauf durchführen zu können.

Im nächsten Abschnitt dieses Artikels werden andere Arbeiten diskutiert, die sich mit der Generierung von Optimierungsproblemen beschäftigen. Außerdem wird dabei die hier eingesetzte Werkzeugsammlung Eli vorgestellt. Im darauf folgenden Abschnitt werden ausgewählte Sprachbestandteile und die dabei auftretenden Typprüfungen vorgestellt. Abschließend wird auf die Übersetzung in ein Optimierungsproblem bzw. Simulationsmodell eingegangen.

2 Verwandte Arbeiten

Die Entwicklung domänenspezifischer Sprachen wird durch diverse Werkzeuge und Programmierrichtlinien unterstützt [7]. In diesem Projekt wird mit der

Werkzeugsammlung Eli gearbeitet, um einen Übersetzer von der Systembeschreibung zum Optimierungs- bzw. Simulationsmodell zu generieren [8]. Eli verfügt über die Möglichkeiten, leicht zu erzeugende Fehlermeldungen ausgeben zu können [3]. So kann beispielsweise angezeigt werden, an welchen Stellen des Pumpensystems ein geforderter Höchstdruck konstruktionsbedingt überschritten wird. Dadurch ist es möglich, gezielt Fehlermeldungen bezüglich der Nichtlösbarkeit des Optimierungsproblems zu erzeugen. Der Anwender sieht so direkt, welche seiner geforderten Systembedingungen oder Verschaltungen zu Widersprüchen führen.

Die grundlegende Formulierung der Optimierungsaufgabe und die Vorstellung der Anwendungsdomäne erfolgt durch unsere Projektpartner in [11]. Dort erfolgt die Modellierung des Problems allerdings noch direkt mit mathematischen Modellierungssprachen wie AMPL [5]. Diese unterstützt zwar schon eine Zusammenfassung von Gleichungen durch Iteration über Mengen, allerdings wird das Optimierungsproblem immer noch mathematisch formuliert und es erfordert ein hohes Verständnis von der Modellierung, falls man diese Umsetzen möchte. Das ausgegebene Ergebnis kann durch die Sprache beeinflusst werden. Somit können hier auch komplexe Ausgaben basierend auf der Lösung angezeigt werden. Hilfreich bei der Modellierung in AMPL ist, dass eine Trennung von dem beschriebenen System mitsamt der geltenden Gleichungen und den tatsächlichen Daten erfolgt.

Dadurch ist es möglich, dass die notwendigen Gleichungen und Ausgabeanweisungen einmalig erstellt werden und anschließend diverse Szenarien, also Lastprofile, umgesetzt werden können. Beim Eingeben der verwendeten Komponenten und Daten müssen die Gleichungen dann nicht nochmal verändert werden. Allerdings ist die Möglichkeit der semantischen Prüfung stark eingeschränkt und Vorabprüfungen auf Lösbarkeit sind quasi nicht vorgesehen. Dies ist dadurch bedingt, dass LP-Löser eine allgemeine Software sind, welche jedes lineare Problem bearbeiten müssen. Lineare Probleme sind eine Menge von uninterpretierten Gleichungen und Ungleichungen, deren Bezug zum abgebildeten Modell erst im Nachhinein erzeugt wird. Betrachtet man nur dieses, so lassen sich schwer Rückschlüsse auf das Pumpensystem herstellen.

Strategien zur Generierung von linearen und nichtlinearen Optimierungsproblemen wurden in [10] untersucht. Hierbei liegt die Formulierung des Ausgangsproblems allerdings noch nah am linearen Programm. Allerdings können damit schon sinnvolle Unterstrukturen gebildet werden, welche die Modellierung vereinfachen und die Modelle lesbarer machen. Aktuellere Vorgehen beinhalten eine Modelltransformation zur Erzeugung von linearen Problemen von abstrakteren Modellen ausgehend [6]. Dabei wurde die selbstständige Laufzeitoptimierung von Programmen untersucht. Aus einer domänenspezifischen Sprache, die die einzelnen Implementierungen beschreibt, wird ein Optimierungsproblem erzeugt. Auch hier erfolgt eine kombinatorische Explosion des Lösungsraumes bei der Verwendung mehrerer Komponenten, was sich in einem Anstieg der benötigten Rechenzeit zum Finden einer Lösung niederschlägt. Eine semantische Vorabprüfung auf die Lösbarkeit des erstellten Optimierungsproblems erfolgt nicht. Die Lösung muss manuell weiterverarbeitet werden.

Eine andere Vorgehensweise der Optimierung von Pumpensystemen kann man z. B. in [1] finden. Hier wird die Simulationssprache modelica derart erweitert, dass der Löser für die Gleichungssysteme der modelica-Umgebung nicht nur das Verhalten simuliert, sondern außerdem entsprechende Parameter optimiert. Der Nachteil hierbei ist, dass nur Parameter optimiert werden, nicht ganze Strukturen von Verschaltungen wie in unserem Projekt. Es muss also direkt bekannt sein, welche Pumpen in welcher Verschaltung verwendet werden. Außerdem ist wieder ein recht hohes Verständnis der Sprache modelica sowie ihrer Erweiterung erforderlich. Auf die Domäne der Pumpensysteme angepasste Fehlermeldungen sind ebenfalls nur schwer möglich, weil modelica für die Modellierung beliebiger hybrider Systeme eingesetzt werden kann. Eine Auswertung der Lösung entfällt, da die berechneten Parameter direkt ein Simulationsmodell bzw. das fertige System ergeben.

Basierend auf den bisher gemachten Erfahrungen wird für das Pumpensystem zunächst ein GMPL-Modell des zugehörigen linearen Optimierungsproblems erstellt. GMPL ist eine Teilsprache von AMPL, welche für unsere Zwecke ausreichend und frei verfügbar ist. Entsprechende Solver findet man z. B. unter [9]. Wir verwenden die Übersetzerbauwerkzeugsammlung Eli, um einen Übersetzer zu entwickeln, der aus unserer domänenspezifischen Sprache ein Optimierungs- und schließlich Simulationsmodell mit den energieoptimalen Pumpen erstellt.

3 Besondere Sprachbestandteile und deren Übersetzung

Im Folgenden werden einige Sprachkonzepte herausgegriffen und näher betrachtet. Dabei wird das Konzept als solches vorgestellt und auf Details bei der Generierung der zugehörigen Optimierungs- und Simulationsmodelle eingegangen.

3.1 Spezifikation von Pumpentypen

Eine Pumpe, hier speziell die Kreiselpumpe, ist eine hydraulische Komponente zum Druckaufbau, welche in Abbildung 3 schematisch dargestellt wird. Sie besteht aus einem Elektromotor, der die über einen Frequenzumrichter aufgenommene elektrische Energie in mechanische Energie als Drehbewegung von Rotorblättern umwandelt. Diese Rotorblätter drücken die Flüssigkeit in Förderrichtung durch die Leitungen, wodurch diese dort unter Druck gesetzt wird. Der eingehende Volumenstrom in $\frac{m^3}{h}$ bleibt erhalten, beeinflusst aber die benötigte Energie, um Druck aufzubauen. Durch eine Regelung der eingehenden elektrischen Leistung über den Frequenzumrichter kann die Rotationsgeschwindigkeit der Rotorblätter eingestellt werden. Die Pumpe erzeugt dann bei kleinerer Stromzufuhr auch weniger Druck.

Die entwickelte domänenspezifische Sprache SHEP ist objektorientiert. Die Typen der Sprache sind Pumpen, Rohre und Armaturen, welche definiert werden können. Ein Beispiel für eine Pumpenspezifikation vom Typ `PumpTypeA` ist im

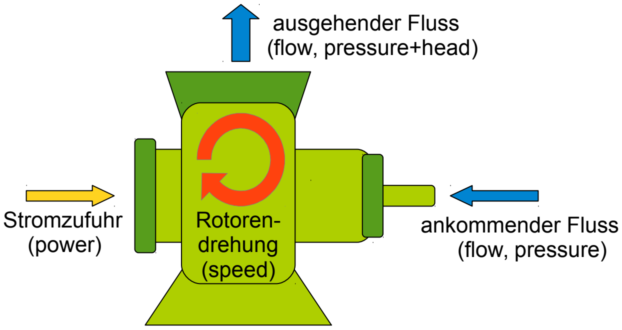


Abb. 3. Schematische Darstellung einer Kreiselpumpe. Durch Zufuhr von elektrischer Energie am Frequenzumrichter bzw. Motor wird eine Rotation ausgelöst, die einen Druck am Ausgang erzeugt und die Flüssigkeit fördert. Durch Regelung der Leistungsaufnahme werden Drehzahl und somit Druckaufbau eingestellt.

Codeausschnitt 1 zu sehen. Zur Spezifikation eines neuen Pumpentyps gehört die Angabe der Kennlinien, eventuelle Kosten der Pumpe sowie die Spezifikation der vorhandenen Anschlüsse der Pumpe.

```
pump PumpTypeA
  characteristic
    speed    head     flow    power;
      400    1.1      0.00     5.00;
      400    1.05     0.78    10.00;
      400    0.985    1.55    15.00;
      400    0.83     2.33    20.00;
      400    0.60     3.10    25.00;
      700    2.94     0.00    20.00;
      700    2.84     1.28    25.00;
      700    2.645    2.55    35.00;
      700    2.24     3.83    50.00;
      700    1.63     5.10    55.00;
   // ... ... ... ...
  costs
    purchase = 100;
  ports
    in   Flange(DN 32, PN 10);
    out  Flange(DN 32, PN 10);
end
```

Codeausschnitt 1. Spezifikation eines Pumpentyps mit dem Namen PumpTypeA, angegebenen Kennlinien, Kosten und den vorhandenen Anschlüssen

Die Kennlinien werden in SHEP als Menge von Messpunkten angegeben. Sie stellen das Verhältnis zwischen Durchfluss, Druckerhöhung, Stromverbrauch und Drehzahl der Rotorblätter dar. Grafisch dargestellt werden diese jeweils in einzelnen Diagrammen, bei denen Kennlinien einer festen Drehzahl gezeichnet werden, die dann den Durchfluss mit der Messgröße vergleichen. Dies wird in Abbildung 4 dargestellt. Bei derartigen Messungen wird die Pumpe bei festen Drehzahlen unterschiedlich großen Volumenströmen ausgesetzt. Dabei misst man den aufgebauten Druck und den Stromverbrauch. Als Einheiten verwenden wir Standardeinheiten.

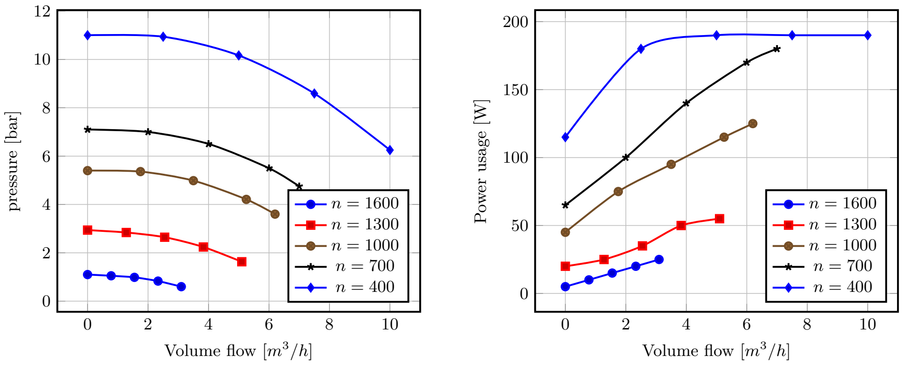


Abb. 4. Beispiel von Pumpenkennlinien in Relation zu einer festen Drehzahl n in Umdrehungen pro Minute.

Bei den Kosten wird der Einkaufspreis bzw. die Installationskosten angegeben. Außerdem können hier andere zeitabhängige laufende Kosten, z. B. Wartung und Stromverbrauch, mitsamt Gewichtung angegeben werden. Diese bilden aufsummiert die ökonomischen Kosten für das System.

Anschlüsse werden mit den Schlüsselwörtern `in` und `out` gekennzeichnet. Der Typ `Flange` steht für die davor definierte Anschlussart Flansch. Dieser hat einen Durchmesser (DN) und einen zulässigen Höchstdruck (PN). Damit kann man die Kompatibilität zu anderen Anschlüssen und zum Gesamtsystem überprüfen.

In Abbildung 5 wird die innerhalb des Übersetzers definierte Klassenhierarchie dargestellt. Jede Klasse verfügt über die dafür typischen Attribute und Gleichungen. Innerhalb der Hierarchie können neue Typen definiert werden. Beim Erben werden dabei die Eigenschaften der übergeordneten Typen bzw. Klassen übernommen. Anschlüsse der Komponente können weiter konkretisiert werden. Die Klassenhierarchie wird dann in ein Teilmengenkonzept der Sprache GMPL überführt. Die entsprechenden Klassen bleiben in ihrer Struktur erhalten.

Der Aufbau des GMPL-Modelles der oben beschriebenen Pumpe wird in Codeausschnitt 2 gezeigt. Den übernommenen Bezeichnern wird ein Unterstrich vorangestellt, um etwaigen Identitäten mit vorhandenen Bezeichnern zu entgehen. Der Unterstrich ist folglich am Anfang eines Bezeichners in SHEP unzulässig. Mit den `set`-Anweisungen wird die Pumpe in die Komponentenhierarchie eingeordnet. Anschließend erfolgen Definitionen für die benötigten Varia-

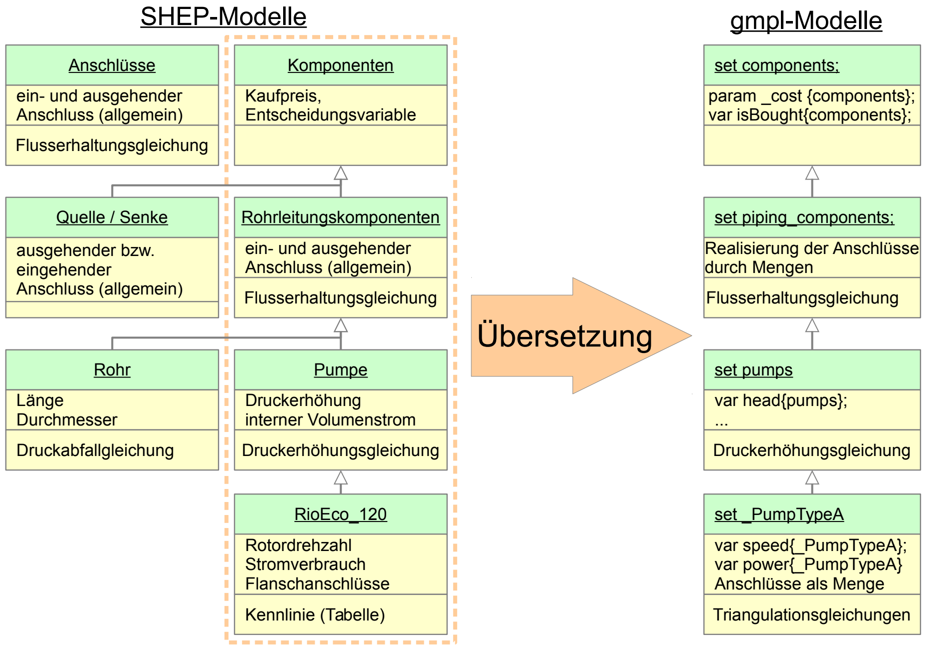


Abb. 5. Übersicht über Hierarchie der in SHEP definierten Komponenten und dem beschriebenen Pumpentyp PumpTypeA. Die dargestellten Einheiten repräsentieren die Klassen mit ihrem Namen, den darin definierten Variablen und den Gleichungen. Die Pfeile repräsentieren die Vererbungsbeziehung zwischen den Klassen.

blen, welche teils für alle Pumpen und teils nur für diesen Pumpentyp angelegt werden. Die Kennlinien werden als zweidimensionale Mengen modelliert, welche jeweils dem Tupel `(flow, head)` entsprechen. Dies ist aufgrund der Beschaffenheit der Kennlinien eindeutig, da bei gleichem Volumenstrom und gleicher Druckdifferenz auch der gleiche Energieverbrauch bzw. Rotorendrehzahl auftreten muss. Diese Werte werden dann den Tupeln zugeordnet. Mit den definierten Variablen und Parametern können schließlich die Gleichungen definiert werden, die für alle Pumpen gleichermaßen gelten.

Die Variablen der Kennlinien werden durch andere Gleichungen miteinander in Beziehung gesetzt. Um diese zu modellieren stehen verschiedene Verfahren mit unterschiedlicher Laufzeit beim anschließenden Lösen des Problems zur Verfügung. Die Wahl des Verfahrens und dessen Parameter werden beim Übersetzungsprozess angegeben. Da es unterschiedlich detaillierte Linearisierungen gibt, kann dies einen Einfluss auf das Ergebnis der Optimierung ausüben.

Bei der Generierung des Simulationsmodells in modelica können die in der Pumpe definierten Attribute als Variablen übernommen werden. Für den Druckaufbau jedoch muss eine Funktion aus den Kennlinien ermittelt werden, abhängig vom Volumenstrom und den zusätzlichen Einträgen. Diese bestimmt dann das simulierte Verhalten der Pumpe. Der Unterschied zum GMPL-Modell ist der,

```
# Definition der Menge mit deklarierten Pumpen
set pumps within piping_components;
set _PumpTypeA within pumps;

# Verwendete Variable
var _power { scenarios, _PumpTypeA };
var _speed { scenarios, _PumpTypeA };
var _head { scenarios, pumps };
var _flow { scenarios, pumps };
var _pressure_port { scenarios, pipe_connectors };
var _flow_port { scenarios, pipe_connectors };

# Definition der Kennlinienparameter
set    basicvalues_PumpTypeA dimen 2; # flow, head
param  basicvalue_PumpTypeA_speed {basicvalues_PumpTypeA};
param  basicvalue_PumpTypeA_power {basicvalues_PumpTypeA};

# Eingabe der Kennlinienwerte
param: basicvalues_PumpTypeA:
           basicvalue_PumpTypeA_speed
           basicvalue_PumpTypeA_power
     := 0.00    1.1      1300      5.00
        0.78    1.05     1300     10.00
# ... ... ... ...

# Gleichungen fuer den Druckaufbau
PressureIncrease {S in scenarios, P in pumps}:
   _pressure_port[S, P, '_out']
    = _pressure_port[S, P, '_in'] + _head[S, P];
```

Codeausschnitt 2. Auszug aus dem erstellten GMPL-Code für den Pumpentyp PumpTypeA mit den Werten der Kennlinie, auftretenden Variablen und Gleichungen. Durch die vordefinierte Menge pumps können Gleichungen vereinfacht werden.

dass man sich bei modelica nicht auf lineare Funktionen beschränken muss, was eventuell eine realitätsnähere Abbildung darstellt. Auch dieses geschieht über externe Funktionen zur Interpolation und auch hier gibt es dafür mehrere Varianten der Umsetzung, die gewählt werden sollten.

3.2 Spezifikation des Systems

Dem Spezifizieren des Systems entspricht das Zeichnen eines Verschaltungsplanes für die Pumpenanlage. Hier werden Komponenten eingebracht, benannt, mit Parametern versehen und verbunden. Schließlich werden Anwendungsfälle spezifiziert, die mit verschiedenen Variablenbelegungen daherkommen. Das Beispielsystem wird in Abbildung 6 gezeigt. Die zugehörige Systembeschreibung findet sich im Codeabschnitt 3.

Zunächst werden die Komponenten definiert. Dabei stehen `Source` und `Sink` für abstrakte Quellen und Senken, um von der konkreten Umgebung unabhängige

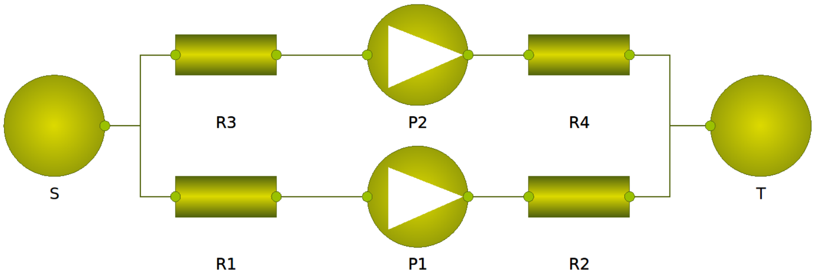


Abb. 6. Beispielsystem aus dem Codebeispiel 3.

```
system MySys
  Source      S;    // Verwendete Komponenten
  PumpTypeA P1;
  PumpTypeA P2 optional;
  SteelPipe  R1, R2, R3 optional, R4 optional;
  Sink        T;
connections         // Verbundene Komponenten
  S  -> R1 -> P1 -> R2 -> T;
  S  -> R3 -> P2 -> R4 -> T;
objective
  minimize costs;   // Zielfunktion (hier: Summe aller Kosten)
scenarios        // Verschiedene Anwendungsfaelle mit Wichtung
  scenario S1 weight 3:
    S.out.pressure   = 3;
    S.out.flow       = 0.78;
    T.in.pressure    = 4.05;
  scenario S2 weight 1:
    S.out.pressure   = 4;
    S.out.flow       = 3.50;
    T.in.pressure    = 8.985;
end
```

Codeausschnitt 3. Konfiguration eines einfachen Systems, bei dem zwischen Quelle und Senke entweder eine oder zwei parallel platzierte Pumpen liegen. Ob beide Pumpen notwendig oder kostengünstiger sind, ergibt sich nach den angegebenen Lastprofilen in den Szenarios.

Teilsysteme zu beschreiben. Dazwischen befindet sich eine Menge von Pumpen und Rohren, von denen einige optional sind, also nicht im optimalen System vorhanden sein müssen. Anschließend werden die Komponenten durch den Pfeil-Operator `->` miteinander verbunden. Dabei wird jeweils der ausgehende Anschluss der linken mit dem eingehenden Anschluss der rechten Komponente verbunden. Diese Werte beeinflussen z. B. auch den möglichen Rohrinnendruck. Daher können sie bei fehlerhafter Zusammenschaltung das Optimierungsproblem unlösbar machen. Eine Analyse der Anschlüsse ermöglicht es, direkt an den auftretenden Stellen Widersprüche kenntlich zu machen.

Die Zielfunktion beschreibt die zu optimierenden Kriterien. In diesem Fall hier werden die Kosten insgesamt minimiert, es werden also die Variablen aus den `costs`-Abschnitten aller Komponenten aufsummiert und diese Summe minimiert. Auch andere Zielkriterien sind hier möglich, müssen dann aber manuell beschrieben werden.

Die Anwendungsfälle darunter beschreiben die notwendigen Anforderungen an das System. Unterschiedliche Szenarien können z. B. Tag- und Nachtbetrieb beschreiben, jeweils mit eigenen Ansprüchen an Druck und Durchfluss im System. Hier kann man durch eine semantische Prüfung fordern, dass die wichtigen Eingangs- und Ausgangsparameter der Quellen und Senken gegeben sind. Wird nämlich der Volumenstrom nicht spezifiziert, so nimmt die Optimierung möglicherweise unsinnige Werte dafür an.

Ein Auszug des erzeugten GMPL-Codes für das Beispielsystem findet sich in Codeausschnitt 4. Komponenten werden als Elemente der entsprechenden Typmengen definiert. Verbindungen können direkt als Tupel von zwei Komponenten und ihren Anschlüssen modelliert werden. Die Gleichungen für die Verbindungen sind für alle Modelle gleich, werden also generiert. Gleichungen der Szenarios können nahezu direkt übernommen werden, müssen nur syntaktisch angepasst werden. Für die Abbildung optionaler Komponenten wird im Optimierungsproblem jeder Komponente die Entscheidungsvariable `_isBought` zugewiesen. Da jede Komponente eine Entscheidungsvariable besitzt, müssen nichtoptionale Komponenten auch als solche gekennzeichnet werden. Die Entscheidungsvariable wird dann `1` gesetzt. Die Zielfunktion wird aus der Angabe in SHEP generiert.

Für alle optionalen Komponenten entscheidet die Optimierung, ob diese verwendet werden oder nicht. Dazu tritt die Entscheidungsvariable in den verschiedenen Gleichungen auf und macht diese ungültig, falls die Komponente nicht benötigt wird. Diese Methoden werden in [11] beschrieben. Bei der Generierung des Optimierungsproblems ist dabei zu beachten, dass solche Entscheidungsvariablen ganzzahlig sind. Dadurch nehmen sie Einfluss auf die Optimierungsgeschwindigkeit. Die Anzahl dieser Entscheidungsvariablen hat also einen wesentlichen Einfluss auf die Verwendbarkeit der domänenspezifischen Sprache und sollte deshalb so gering wie möglich gehalten werden.

4 Generierung effizienter Optimierungsprobleme

Eine Optimierungsmöglichkeit bietet die Elimination von Entscheidungsvariablen für Komponenten. Im obigen Beispiel kann man die Entscheidungsvariable für P2, R3 und R4 zu einer zusammenfassen, da es sich dabei um einen einzigen Pfad handelt. Die Pfade können durch ähnliche Verfahren wie bei der Programmanalyse ermittelt werden. Alternativ können auch Pfade, welche einen anliegenden Druck nicht unterstützen, gänzlich weggelassen werden.

Ein weiterer wesentlicher Punkt ist die Linearisierung nichtlinearer Ausdrücke. In der Pumpendomäne gibt es nativ diverse nichtlineare Zusammenhänge, z. B. beim Druckabfall. Werden die entsprechenden Gleichungen direkt in das

```
# Komponenten des Systems als Elemente der entsprechenden Typmengen
set _PumpTypeA within pumps := { '_P1' };
set _SteelPipe within pipes := { '_R1', '_R2', '_R3', '_R4' };
set Source := { '_S' };
set Sink := { '_T' };

# Verbindungen des Systems als Tupelmenge
set connections within pipe_connectors cross pipe_connectors
  := { ('_S',  '_out', '_R1', '_in'),
       ('_R1', '_out', '_P1', '_in'), ...  };

# Lastfälle (Szenarien) des Systems
set scenarios := { '_S1', '_S2' };

# Druckerhaltungsgleichung für die Verbindungen
PressurePreservationConnections {S in scenarios, (U, P) in
    pipe_connectors, (U, P, V, Q) in connections}:
  _pressure_port[S, U, P] = _pressure_port[S, V, Q];

# Übernommene Gleichungen der Lastfälle
equation_S1_1: _pressure_port['_S1', '_S', '_out'] = 3;
equation_S1_2: _flow_port['_S1', '_S', '_out'] = 0.78;
  ...
# Abschalten nicht benötigter Entscheidungsvariablen
_S_isBought: _isBought['_S'] = 1;
_P1_isBought: _isBought['_P1'] = 1;
...
# Zielfunktion (aufsummierte Kosten für die gekauften Komponenten)
minimize objective: sum{C in components} _purchase[C] *
    _isBought[C];
```

Codeausschnitt 4. Auszug aus dem erstellten GMPL-Code für das im Codeausschnitt 3 vorgestellte System. Durch die vordefinierte Menge pumps können Gleichungen vereinfacht werden.

Optimierungsproblem übernommen, so ist auch das ganze Optimierungsproblem nichtlinear. Dieses kann dann nicht mehr so schnell und zuverlässig gelöst werden wie ein lineares Optimierungsproblem [2]. Daher verwenden wir Linearisierungstechniken wie Interpolation und Substitution, um die nichtlinearen Ausdrücke umzuformen.

Dazu zählen zunächst einfache Möglichkeiten des Übersetzerbaus wie die Konstantenfaltung oder das Ausmultiplizieren von Ausdrücken. Bleiben nichtlineare Terme zurück, muss man diese durch spezielle mathematische Verfahren annähern. Die einfachste Möglichkeit ist dabei, die Terme stückchenweise durch lineare Funktionen zu interpolieren. Position und Anzahl der Stützstellen haben entscheidenden Einfluss auf die Genauigkeit der Näherung. Aufgrund dieser Abrundungen muss das Ergebnis der Optimierung zum Schluss gegen die Ausgangsgleichungen getestet werden. So kann sichergestellt werden, dass das Ergebnis der Optimierung verwertbar ist.

5 Zusammenfassung und Ausblick

Dieser Artikel beschreibt eine domänenspezifische Sprache zur Planung optimaler Pumpensysteme. Entwickelt wurde die Sprache mit dem Übersetzerbautool Eli. Oft wiederkehrende Bauteile, wie Pumpen und Rohre, können schnell mit ihren jeweiligen Eigenschaften beschrieben werden. Die in ihnen geltenden Verhältnisse in Form von Gleichungen und Variablen sind bereits in den Sprachkonzepten verankert. Dadurch braucht man bei der Spezifikation einer neuen Pumpe nur die jeweiligen Anschlüsse und Pumpenkennlinien angeben und kann sie anschließend für den Systemaufbau verwenden. Dieser funktioniert ähnlich einer Programmiersprache, indem Komponenten deklariert werden können. Anschließend kann man diese miteinander verbinden, um ein Pumpensystem aufzubauen.

Dabei geschehen ständig Prüfungen durch die syntaktische und semantische Analyse. Darunter fallen Abfragen, welche Komponenten mit ihren Anschlüssen überhaupt verbunden werden können und welche Variablen innerhalb einer Komponente bekannt sind. Eine besondere semantische Analyse stellt hier die Vorabprüfung des zu erstellenden Optimierungsproblems dar. Hierbei können Methoden des Übersetzerbaus und der Programmanalyse verwendet werden. Dabei werden die Grenzen des Flüssigkeitsflusses der Anlage iterativ ausgelotet. Die Analyse bringt den Vorteil, dass für den Nutzer lesbare Fehlermeldungen erzeugt werden können. Würde das Optimierungsproblem direkt von einem LP-Solver bearbeitet werden, dann würde man für nicht lösbare Instanzen kaum Informationen erhalten und wenn dann nur über generierte Variable und Gleichungen.

Bei den auftretenden Linearisierungen für die Kennlinien und nichtlinearen Ausdrücke entstehen Rundungsfehler. Daher wird zusätzlich zum Optimierungsmodell auch ein Simulationsmodell erstellt, welches das modellierte System überprüfen soll. In der Simulation werden dann die optimierten Parameter eingestellt und die auftretenden Drücke und Durchflüsse getestet. Sollten hier zu große Abweichungen auftreten, muss die Optimierung mit genaueren Linearisierungen wiederholt werden.

Die Sprache wird demnächst um neue Eingabemethoden von Kennlinien erweitert. Diese können dann z. B. aus CSV-Dateien geladen werden, was zu einer erhöhten Übersicht der zu schreibenden Dateien führt. Auch ist es vorstellbar, die Kennlinien direkt als Funktion anzugeben und Stützstellen selbst zu bestimmen. Dadurch lassen sich dann auch theoretische oder noch konstruierte Pumpen ohne konkrete Messwerte simulieren und analysieren.

In weiteren Schritten sollen dann die Pumpen detaillierter modelliert werden. Diese bestehen dann aus mehreren Bauteilen, wie Frequenzumrichter und Motor, welche zusätzliche Einstellungsparameter ermöglichen. Dadurch ist eine weiterführende Optimierung möglich. Auch andere Bauteile erhalten zusätzliche Parameter, wie z. B. die Beschaffenheit des Rohres, welche Einfluss auf den darin geltenden Druckabfall nimmt. Vorgefertigte Komponenten, wie z. B. spezielle Pumpen, werden durch Bibliotheken modularisiert zugreifbar gemacht.

Literatur

[1] Johan Åkesson u. a. „Modeling and optimization with Optimica and JModelica. Languages and tools for solving large-scale dynamic optimization problems“. In: *Computers & Chemical Engineering* 34.11 (2010), S. 1737–1749.

[2] Pietro Belotti u. a. „Mixed-integer nonlinear optimization“. In: *Acta Numerica* 22 (2013), S. 1–131.

[3] Christian Berg und Wolf Zimmermann. „Evaluierung von Möglichkeiten zur Implementierung von Semantischen Analysen für Domänenspezifische Sprachen.“ In: *Software Engineering (Workshops)*. 2014, S. 111–128.

[4] Deutsche Energie-Agentur GmbH (dena). „Energiesparpaket: Leuchtturmprojekt zur energetischen Optimierung von Pumpensystemen“. In: (2011).

[5] Robert Fourer, David Gay und Brian Kernighan. *Ampl*. Bd. 119. Boyd & Fraser, 1993.

[6] S Götz u. a. „Modeldriven self-optimization using integer linear programming and pseudoboolean optimization“. In: *Proceedings of ADAPTIVE* (2013), S. 55–64.

[7] Gabor Karsai u. a. „Design guidelines for domain specific languages“. In: *arXiv preprint arXiv:1409.2378* (2014).

[8] Uwe Kastens, Peter Pfahler und Matthias Jung. „The eli system“. In: *Compiler Construction*. Springer. 1998, S. 294–297.

[9] Andrew Makhorin. *GLPK - GNU Project - Free Software Foundation (FSF)*. 2015. URL: `http://www.gnu.org/software/glpk/`.

[10] Frederic H Murphy und Edward A Stohr. „An intelligent system for formulating linear programs“. In: *Decision Support Systems* 2.1 (1986), S. 39–47.

[11] Peter Pelz u. a. *Designing Pump Systems by Discrete Mathematical Topology Optimization: The Artificial Fluid Systems Designer*. Techn. Ber. International Rotating Equipment Conference, 2012.

[12] Bundesministerium für Wirtschaft und Energie. *Gesamtausgabe der Grafiken zu Energiedaten*. 26.08.2015. URL: `http://www.bmwi.de/BMWi/Redaktion/PDF/E/energiestatistiken-grafiken,property=pdf,bereich=bmwi2012,sprache=de,rwb=true.pdf`.

Danksagung

Dieses Projekt wird gefördert durch das Bundesministerium für Wirtschaft und Technologie unter dem Förderkennzeichen 03ET1134C. Partner in diesem Projekt sind die Technische Universität Darmstadt und die KSB Aktiengesellschaft. Dank gilt außerdem Felix Knispel für die Zuarbeit zum Projekt und zu diesem Artikel.

Evaluating Interpreter Design in Prolog

Philipp Körner, David Schneider and Michael Leuschel

Institut für Informatik, Heinrich Heine University Düsseldorf, Germany
{p.koerner, david.schneider}@hhu.de, leuschel@cs.uni-duesseldorf.de

Abstract. The semantics and the recursive execution model of Prolog make it very natural to express language interpreters in form of AST (Abstract Syntax Tree) interpreters where the execution follows the tree representation of a program. An alternative implementation technique is that of bytecode interpreters. These interpreters transform the program into a compact and linear representation before evaluating it and are generally considered to be faster and to make better use of resources.
In this paper, we discuss different ways to express the control flow of interpreters in Prolog and present several implementations of AST and bytecode interpreters.
On a simple language designed for this purpose, we evaluate whether techniques best known from imperative languages are applicable in Prolog and how well they perform. Our ultimate goal is to assess which interpreter design in Prolog is the most efficient as we intend to apply these results to a more complex language. However, we believe the analysis in this paper to be of more general interest.

1 Introduction

Writing simple language interpreters in Prolog is pretty straightforward. The data structures and language semantics are a natural match to the evaluatation of programs, in particular if those are represented as trees. Selecting which predicate to execute in order to evaluate a part of a program is done by unifying the part of the program to be executed next with the set of rules in Prolog's database that implement the language semantics. Subsequent execution steps can be chosen using logic variables that are bound to substructures of the matched node.

Although this approach to interpreter construction is a natural match to Prolog, the question remains if it is the most efficient way to implement the instruction dispatching logic for any language implemented in Prolog. In particular, we have developed such an interpreter [4] for the full B language and wanted to evaluate the potential for improving its performance, by using alternate implementation techniques.

Interpreters implemented in imperative languages, especially low-level languages, often make use of alternative techniques for implementing the dispatching logic, taking advantage of available data structures and programming paradigms that might be available in higher-level languages.

In this article, we try to explore if some of these techniques can be implemented in Prolog or applied in interaction with a Prolog runtime with the goal to assess if the instruction dispatching for language interpreters can be made faster while keeping the language semantics in Prolog.

In order to examine the performance of different dispatching models in Prolog, we have defined a simple imperative language named Acol, which is described in section 2. For Acol we have created several implementations described in section 3, that use different paradigms for the dispatching logic. Finally, in section 4, we present a set of benchmarks written in Acol used to evaluate the implemented interpreters when running on SICStus and SWI Prolog.

2 A Simple Language

As a means to evaluate the different interpreter designs described in section 3, we have defined a very simple and limited language named Acol[1].

Acol is an imperative language consisting of three kinds of statements: while-loops, if-then-else statements and variable assignments. The only supported value type is integer. Furthermore, Acol offers a few arithmetic operators (addition, subtraction, multiplication and modulo), comparisons (less than (or equal to), greater than (or equal to) and equals), as well as a boolean `not` operator.

A simple Acol program is shown in fig. 1.

```
# the initial environment (i.e. input):
# base = 2
# exponent = 5

# the program
val = 1;
while exponent > 0 {
    val = val * base;
    exponent = exponent - 1;
}
```

Fig. 1: A small program

3 Interpreter Implementations

There are many ways to implement Acol, in C as well as in Prolog. Considering several different interpreter implementation techniques, in this section we will describe possible designs of interpreters and the closely related representations of the Acol programs. The interpreters are based on either traversing the abstract syntax tree representation of a program or on compiling the program to bytecode first and evaluating this more compact representation instead.

All interpreters share the same implementation of the language semantics exposed by an object-space API [5]. In order to keep the implementations

[1] Acol is not a backronym for Acol is a computable language

simple and compatible, they all call into the same object space. Nonetheless, the interpreters differ very much in the representation of the program and, hence, in the process of dispatching.

In order to discuss the differences, we will translate a small example program shown in fig. 1 into the different representations and show an excerpt of the interpretation logic for each paradigm. In fig. 2, the AST for the example program is depicted.

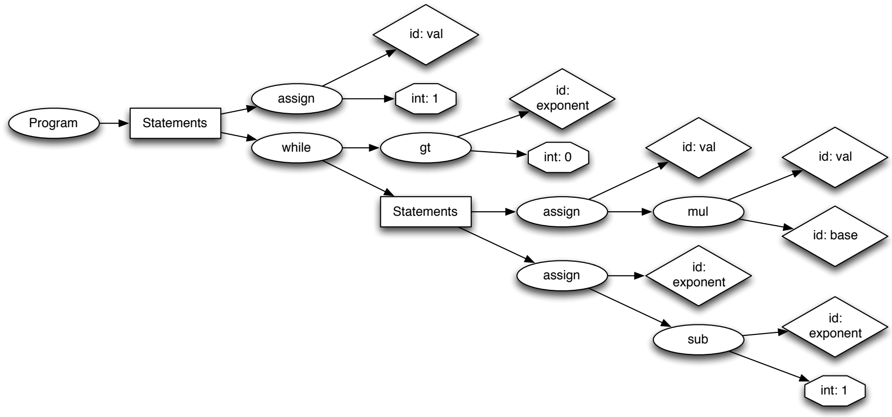


Fig. 2: AST

3.1 AST Interpreter

The most natural way to implement an interpreter in Prolog is in form of an AST-interpreter since it synergises very well with its execution model.

The data structure used for this interpreter is the tree representation of the program as generated by the parser, the AST (abstract syntax tree). In Prolog, the AST can be represented as a single term as shown in fig. 3. The program itself is a Prolog list of statements. However, every statement is represented as its own tree. Block statements, i.e. `if` and `while`, will contain a list of statements themselves.

```
[assign(id(val), int(1)),
 while(gt(id(exponent), int(0)),
        [assign(id(val), mul(id(val), id(base))),
         assign(id(exponent), sub(id(exponent), int(1)))])]
```

Fig. 3: Prolog representation of the AST

```
ast_int([], Env, _Objspace, Env).
ast_int([H|T], EnvIn, Objspace, EnvOut) :-
    ast_int(H, EnvIn, Objspace, Env), ast_int(T, Env, Objspace, EnvOut).
ast_int(if(Cond, Then, Else), EnvIn, Objspace, EnvOut) :-
    eval(Cond, EnvIn, Objspace, X),
    (X == true -> ast_int(Then, EnvIn, Objspace, EnvOut)
               ;  ast_int(Else, EnvIn, Objspace, EnvOut)).
ast_int(assign(id(Var), Expr), EnvIn, Objspace, EnvOut) :-
    eval(Expr, EnvIn, Objspace, Res), Objspace:store(EnvIn, Var, Res, EnvOut).
ast_int(while(Cond, Instr, _Invariant, _Variant), EnvIn, Objspace, EnvOut) :-
    ast_while(Cond, Instr, EnvIn, Objspace, EnvOut).
```

Fig. 4: Dispatching in a Prolog AST interpreter

The AST interpreter will examine the first element of the list, execute this statement and continue with the rest of the list, as can be seen in fig. 4. Every tree encountered this way is evaluated recursively.

Choosing the implementation for each node in the tree is done by unifying the current root node with the set of evaluation rules. This approach benefits from the first argument indexing [7] optimisation done by most Prolog systems.

3.2 Bytecode Interpreters

We have defined a simple set of bytecodes, described below, as a compilation target for Acol programs. Based on these instructions we will introduce a series of bytecode-interpreters that explore different implementation approaches in Prolog and C.

As many bytecode interpreters for other languages, ours are *stack-based*. Some opcodes may create or load objects and store them on the evaluation stack, e.g. push or load. Yet others may in turn consume objects from the stack and create a new one in return, e.g. add. Lastly, a single opcode is used to manipulate the environment, i.e. assign. An exhaustive list is shown in table 1.

Imperative Bytecode Interpreter Usually, bytecode interpreters are written in imperative languages, that are rather low-level, e.g. C, that allow more control about how objects are laid out in memory and provide fine grained control over the flow of execution.

To introduce the concept of a bytecode interpreter, we present an implementation of Acol beyond Prolog, that is purely written in C.

The bytecode is stored as a block of memory, that can be interpreted as an array of bytes. The index of this array that should be interpreted next is called the program counter. After that opcode is executed, the program counter is incremented by one plus the size of its arguments. However, it may be set to an arbitrary index by opcodes implementing jumps. Integer arguments are encoded in reverse byte order. An example for a bytecode based on the program above is shown in fig. 5.

#	Name	Arguments	Semantics
10	jump	4 bytes encoded PC	jumps to new PC
11	jump-if-false	4 bytes encoded PC	jumps to new PC if top element is falsey
12	jump-if-true	4 bytes encoded PC	jumps to new PC if top element is truthy
20	push1	1 byte encoded integer	push the argument on the stack
21	push4	4 bytes encoded integer	push the argument on the stack
40	load	4 bytes encoded variable ID	push variable on the stack
45	assign	4 bytes encoded variable ID	store top of the stack in variable
197	mod	-	pop operands, push result of operation
198	mul	-	pop operands, push result of operation
199	sub	-	pop operands, push result of operation
200	add	-	pop operands, push result of operation
240	not	-	pop operand, push negation
251	eq	-	pop operands, push result of comparison
252	le	-	pop operands, push result of comparison
253	lt	-	pop operands, push result of comparison
254	ge	-	pop operands, push result of comparison
255	gt	-	pop operands, push result of comparison

Table 1: A bytecode for the described language

```
unsigned
char bc[] = {20, 1,           // push integer 1 on the stack
             45, 2, 0, 0, 0,  // store it in variable at index 2
                              // (i.e. val)
             40, 1, 0, 0, 0,  // load the variable at index 1
                              // (i.e. exponent)
             20, 0,           // push 0
             255,             // greater than
             11, 54, 0, 0, 0, // jump behind loop
                              // if condition is falsey
             40, 2, 0, 0, 0,  // load val
             40, 0, 0, 0, 0,  // load base
             198,             // mul
             45, 2, 0, 0, 0,  // store val
             40, 1, 0, 0, 0,  // load exponent
             20, 1,           // push 1
             199,             // sub
             45, 1, 0, 0, 0,  // store exponent
             10, 7, 0, 0, 0,  // jump to beginning of loop
             0}               // terminate
```

Fig. 5: Example bytecode in C

```
while (pc < bc_len) {
    unsigned char *arg = bc + pc + 1;
    switch (bc[pc]) {
        case JUMP:
            pc = decode_arg4(arg); break;
        case LOAD:
            index = decode_arg4(arg);
            push(stack, env[index]);
            pc += 5; break;
        case ASSIGN:
            env[arg] = pop(stack);
            pc += 5; break;
        case ADD:
            b = pop(stack);
            a = pop(stack);
            push(stack, add(a, b));
            pc++; break;
        // ... many further cases
    }
}
```

Fig. 6: Dispatching logic in C

```
while (pc < bc_len) {
    unsigned char *arg = bc + pc + 1;
    switch (bc[pc]) {
        case JUMP:
            pc = decode_arg4(arg); break;
        case LOAD:
            index = decode_arg4(arg)
            push(stack, env[index]);
            pc += 5; break;
        case ASSIGN:
            index = decode_arg4(arg);
            PL_put_term(env[index], pop(s));
            pc += 5; break;
        case ADD:
            arg1 = PL_new_term_refs(3);
            arg2 = arg1 + 1;
            var = arg1 + 2;
            PL_put_term(arg2, pop(s));
            PL_put_term(arg1, pop(s));
            PL_call_predicate(NULL,
                              PL_Q_NORMAL,
                              predicate_add,
                              arg1);
            push(s, var);
            pc++; break;
        // ...  many further cases
    }
}
```

Fig. 7: Dispatching logic using SWI's C-Interface

The dispatching logic is implemented as a switch-statement, that is contained in a loop. An excerpt of the implementation of our bytecode-interpreter in C is shown in fig. 6. Every case block contains an implementation of that specific opcode. After the opcode is executed, the program counter is advanced or reset and the next iteration of the main loop is commenced.

C-Interfaces We made the digression into an interpreter written in C not only to present the concept of bytecode interpreters. Instead, we can utilise the same dispatching logic, but instead of calling an object space that is implemented in C, we can use the C interfaces provided by the Prolog runtimes we consider (SICStus and SWI) to call arbitrary Prolog predicates. This way, we can query the aforementioned object space that contains the semantics of ACOL, but is implemented in Prolog. An excerpt when using the C interface of SWI Prolog is shown in fig. 7.

Then, the main loop dispatches in C, but the objects on the evaluation stack are created and the operations are executed by Prolog predicates.

Prolog Facts The main issue with bytecode interpreters in Prolog is to efficiently implement jumps to other parts of the bytecode. With an interpreter in C, all we have to do is re-assigning the program counter variable. Prolog, however, does not offer arrays with constant-time indexing.

The idiomatic way to simulate an array would be to use a Prolog list, but on this data structure we can perform lookups only in $\mathcal{O}(n)$. However, there are other representations of the program that allow jumping to another position faster.

One way to express such a lookup in $\mathcal{O}(1)$ is to transform the bytecode into Prolog terms `bytecode(ProgramCounter, Instruction, Arguments)`. Those terms are written into a seperate Prolog module that is loaded afterwards. The first argument indexing optimisation then allows performing lookups in constant time.

In contrast to an interpreter written in C, it does not perform well to encode integer arguments into reverse byte-order arguments. Instead, we use the Prolog primitives, i.e. integers for values and atoms for variable identifiers.

```
bytecode(0, 20, 1).
bytecode(2, 45, val).
bytecode(7, 40, exponent).
bytecode(12, 20, 0).
bytecode(14, 255, []).
bytecode(15, 11, 55).
bytecode(20, 40, val).
bytecode(25, 40, base).
bytecode(30, 198, []).
bytecode(31, 45, val).
bytecode(36, 40, exponent).
bytecode(41, 20, 1).
bytecode(43, 199, []).
bytecode(44, 45, exponent).
bytecode(49, 10, 7).
bytecode(54, 0, []).
```

Fig. 8: Bytecode as Prolog facts

Figure 8 shows a module that is generated from the bytecode. The interpreter fetches the instruction located at the current program counter, executes it and increments the program counter accordingly. This is repeated until it encounters a special zero instruction that denotes the end of the bytecode.

The dispatching mechanism is shown in fig. 9. Similar to an interpreter in C, every opcode has an implementation in Prolog that calls into the object space. Any rule of `fact_int` is equivalent to a `case` statement in C.

Sub-Bytecodes Another design is based on the idea that a program is executed *block-wise*, i.e. a series of instructions that is guarenteed to be executed in this specific order. This is very simple since ACOL does not include a goto-statement that allows arbitrary jumps. From a programmer's point of view, blocks are the body of while-loops or those of if-then-else statements.

Instead of linearising the entire bytecode, only a block is linearised at once. In order to deal with blocks that are contained by another block (e.g. nested loops), two special opcodes are added. They are used to suspend the execution

```
fact_int(PC, Objspace, Env, Stack, REnv) :-
    generated:bc(PC, Instr, Args), % fetch the instruction
    fact_int(Instr, Args, PC, Stack, Env, Objspace, REnv).
fact_int(200, _Args, PC, [Y, X|Stack], Env, Objspace, REnv) :-
    Objspace:add(X, Y, Res), NewPC is PC + 1,
    fact_int(NewPC, Objspace, Env, [Res|Stack], REnv).
% fact_int also has implementations of all the other bytecodes...
```

Fig. 9: Dispatching in the facts-based interpreter

of the current block and look up the *sub-bytecodes* of the contained blocks that are referenced via its arguments. After those sub-bytecodes are executed, the execution of the previous bytecode is resumed.

The special if-opcode references the blocks of the corresponding then- and else- branches. After the condition is evaluated, only the required block is looked up and executed. The other special opcode for while-loops references the bytecode of the condition that is expected to leave true or false on the stack, as well as the body of the loop. The blocks corresponding to condition and body are evaluated in turn until the condition does not hold any more, so the execution of its parent block can continue.

Similar to the facts in the interpreter above, the sub-bytecodes are asserted into their own module to allow fast lookups.

Figure 10 shows an example that includes the special opcode for the while-statement.

```
[20, 1, 45, val, % val = 1
 2, 0, 1]        % while (condition encoded in sub-bytecode 0,
                 %        body encoded in sub-bytecode 1)

% Sub-bytecodes
sbc(0, [40, exponent, 20, 0, 255]).
sbc(1, [40, val, 40, base, 198, 45, val, 40, exponent, 20, 1, 199]).
```

Fig. 10: Bytecode with sub-bytecodes

Figure 11 shows an excerpt of the dispatching logic used for this interpreter. The recursion in `bc_int2` will update the bytecode-list with its tail instead of manipulating a program counter. Hence, in this implementation, the interpreter can only move forward inside of a block. If it is required to move backwards in the program, it is only possible to re-start at the beginning of a block.

3.3 Rational Trees

Based on [1], we have created implementations of an AST- and a bytecode-interpreter for Acol that use the idea of rational trees to represent the program

```
bc_int([], Env, Stack, _Objspace, Env, Stack).
bc_int([H|R], Env, Stack, Objspace, REnv, RStack) :-
    bc_int2(H,R, Env, Stack, Objspace, REnv, RStack).
% special bytecodes for evaluating blocks of an if-statement
bc_int2(1, [T, E|R], Env, [Cond|Stack], Objspace, REnv, RStack) :-
    (Cond == true -> subbytecodes:sbc(T, Then),
                     h_bc_int(Then, [], Env, Objspace, TEnv)
                  ;  subbytecodes:sbc(E, Else),
                     h_bc_int(Else, [], Env, Objspace, TEnv)),!,
    bc_int(R, TEnv, Stack, Objspace, REnv, RStack).
% special bytecodes for evaluating blocks of a while-loop
bc_int2(2, [C, I|R], Env, Stack, Objspace, REnv, RStack) :-
    subbytecodes:sbc(C, Cond),
    bc_int(Cond, Env, [], Objspace, Env, [Res]),
    (Res == true -> subbytecodes:sbc(I, Instr),
                    h_bc_int(Instr, [], Env, Objspace, T),!,
                    bc_int2(2, [C, I|R], T, Stack, Objspace, REnv, RStack)
                 ;  !, bc_int(R, Env, Stack, Objspace, REnv, RStack)).

bc_int2(200, R, Env, [Y, X|Stack], Objspace, REnv, RStack) :-
    Objspace:add(X, Y, Res),!,
    bc_int(R, Env, [Res|Stack], Objspace, REnv, RStack).
% bc_int2 also has implementations of all the other bytecodes...
```

Fig. 11: Dispatching on bytecodes with sub-bytecodes

being evaluated. This technique aims to improve the performance of jumps by using recursive data structures containing references to the following instructions.

AST-Interpreter with Rational Trees Since ACOL does not include a concept of arbitrary jumps as used in [1], it is not possible to achieve the speed-up described in the referenced paper. However, we can make use of the basic idea for the representation of programs: every statement has a pointer to its successor statement.

In our naive AST interpreter, a new Prolog stack frame is used for every level of nested loops and if-statements. Instead of returning from each evaluation to the predicate that dispatched to the sub-statement, we can make use of Prolog's tail-recursion optimisation and continue with the next statements directly.

```
assign(id(val), int(1),
  while(gt(id(exponent), int(0)),
    assign(id(val), mul(id(val), id(base)),
      assign(id(exponent), sub(id(exponent), int(1)),
        while(gt(id(exponent), int(0)),
         ...)))
    end))
```

Fig. 12: Rational tree representation

For our example program, we generate an infinite data structure for the while-loop depicted in fig. 12. The concept of rational trees allows us to have the while-term re-appearing in its own body, so it has not to be saved in a stack frame.

The last statement end is artificially added to indicate the end of the program so that the interpreter may halt.

Then, the dispatching logic is still very similar to the naive AST interpreter as shown in fig. 13.

```
rt_int(end, Env, _, Env) :- !.
rt_int(assign(id(Var), Expr, Next), Env, Objspace, REnv) :-
    eval(Expr, Env, Objspace, Res),
    Objspace:store(Env, Var, Res, EnvOut), !,
    rt_int(Next, EnvOut, Objspace, REnv).
rt_int(if(Cond, Then, Else), Env, Objspace, REnv) :-
    eval(Cond, Env, Objspace, V),
    (V == true -> !, rt_int(Then, Env, Objspace, REnv)
                ; !, rt_int(Else, Env, Objspace, REnv)).
rt_int(while(Cond, Instrs, Else), Env, Objspace, REnv) :-
    eval(Cond, Env, Objspace, V),
    (V == true -> !, rt_int(Instrs, Env, Objspace, REnv)
                ; !, rt_int(Else, Env, Objspace, REnv)).
```

Fig. 13: Dispatching in a rational tree interpreter

Bytecode-Interpreter With Rational Trees In Prolog, Rational trees can also be used for bytecodes. Jumps are removed from that representation entirely. While-loops are unrolled into an infinite amount of alternated bytecodes of the condition and if-statements that contain the body of the loop in their then-branch and the next statement after the loop in their else-branch. An example is shown in fig. 14.

At first glance, it looks weird that the opcode integers are replaced by human-readable descriptions. However, functors are limited to atoms and then there is not much difference between atoms that contain only a number or short readable names. We chose the latter one because they are by far more comprehensible.

```
push(1, assign(val,                                  % code before the loop
  load(exponent, push(0, gt(                         % condition (1)
    if(load(val, load(base, mul(store(val,           % while-body (1)
         load(exponent, push(1, sub(store(exponent,  % while-body (1)
           load(exponent, push(0, gt(                % condition (2)
             if(load(val, load(base(, ....))),       % while-body (2)
               end)))))))))))                        % end of while (2)
       end))))))                                     % end of while (1)
```

Fig. 14: Bytecode with rational trees

```
rt_bc_int(end, Env, Stack, _Objspace, Env, Stack).
rt_bc_int(if(Then, Else), Env, [X|Stack], Objspace, REnv, RStack) :-
    (X == true -> !, rt_bc_int(Then, Env, Stack, Objspace, REnv, RStack)
               ; !, rt_bc_int(Else, Env, Stack, Objspace, REnv, RStack)).
rt_bc_int(push(Arg, Next), Env, Stack, Objspace, REnv, RStack) :-
    Objspace:create_integer(Arg, Val),!,
    rt_bc_int(Next, Env, [Val|Stack], Objspace, REnv, RStack).
rt_bc_int(load(Arg, Next), Env, Stack, Objspace, REnv, RStack) :-
    Objspace:lookup(Arg, Env, Val), !,
    rt_bc_int(Next, Env, [Val|Stack], Objspace, REnv, RStack).
rt_bc_int(add(Next), Env, [Y, X|Stack], Objspace, REnv, RStack) :-
    Objspace:add(X, Y, Res), !,
    rt_bc_int(Next, Env, [Res|Stack], Objspace, REnv, RStack).
% rt_bc_int implements all other opcodes as well...
```

Fig. 15: Dispatching in a bytecode interpreter with rational trees

The dispatching is pretty similar to the AST interpreter that utilises rational trees, as shown in fig. 15. The main difference between those two interpreters is that this one uses a simulated stack to evaluate terms instead of Prolog's call stack.

4 Evaluation

To compare the performance of the different interpreters for Acol, we selected a set of different benchmarks. Because the language is very limited, it is hard to design "real-world programs".

In this section, we present those benchmarks and compare their results. Each program was executed with every interpreter ten times. The runtime consists only of the time spent in the interpreter, the compilation time is excluded.

The benchmarks were run on a machine that runs a linux with a 3.19.0-25-generic 64-bit kernel on an Intel i5-2400 CPU @ 3.10GHz. Two Prolog implementations were considered: SICStus Prolog 4.3.2, a commercial product, and SWI Prolog 7.2.2, a free open-source implementation. All C code was compiled by gcc 4.9.2. with the `-O3`-flag.

Since Acol does not offer complex features, we expect that the dispatching claims a bigger share of the runtime than the actual operations.

4.1 Benchmarks

Prime Tester The first benchmark is a naive prime tester. The program is depicted in fig. 16. The environment was pre-initialised with `is_prime` $:= 1$, `start` $:= 2$, and `V` $:= 34\,265\,341$.

Fibonacci Another benchmark is the calculation of the fibonacci sequence. However, we expect that most of the execution time will consist of the addition and subtraction of two big numbers and that the interpreter overhead itself is rather

```
while (start < V) {
    if (V mod start == 0) {
        is_prime := 0;
    } else {
        is_prime := is_prime;
    }
    start := start + 1;
}
```

Fig. 16: Prime Tester Program

```
i := 1;                     i := 1;
while i < n {               while i < n {
    b := b + a;                 b := b + a mod 1000000;
    a := b - a;                 a := b - a mod 1000000;
    i := i + 1;                 i := i + 1;
}                           }
```

Fig. 17: Fibonacci Programs

small. Therefore, a second version that calculates the sequence modulo 1 000 000 is included.

Again, the environment is pre-initialised, in this case with `a` := 0, `b` := 1 and `n` := 400 000. To ensure a significant runtime for the second version, the input is modified so it calculates a longer sequence, i.e. `n` := 10 000 000.

Generated ASTs Lastly, some programs were generated pseudo-randomly. Such a generated AST consists of 20 to 50 statements that are uniformly chosen from while-loops, if-statements and assignments. The body of a loop and both branches of if-statements also consist of 20 to 50 statements. However, if the nesting exceeds a certain depth, only assignments are generated for this block.

In order to guarentee termination, while-loops are always executed 20 times. An assignment is artifically inserted before the loop that resets a loop counter, as well as another assignment that increments this variable at the beginning of the loop.

For assignments and if-conditions, a small subtree is generated. The generator chooses uniformly between five predetermined identifiers, constants ranging from -1 to 3, as well as additions and subtractions. If-conditions have to include exactly one comparison operator.

The generator does include neither multiplications, because they caused very large integers that slowed down the Prolog execution time significantly, nor modulo operations, to avoid division by zero errors.

Three different benchmarks were generated using arbitrary seeds. Their purpose is to complement the other three handwritten benchmarks, which are rather small and might benefit from caching of the entire AST.

Benchmark	Interpreter	SICSTus	SWI
Prime Tester	AST	66.46 ± 0.32 (1.00)	438.05 ± 6.32 (1.00)
	Sub-Bytecodes	85.11 ± 0.45 (1.28)	565.73 ± 27.28 (1.29)
	Facts	96.77 ± 1.82 (1.46)	537.64 ± 18.03 (1.23)
	C-Interface	183.58 ± 2.85 (2.76)	82.11 ± 1.77 (0.19)
	AST w/ Rational Trees	67.21 ± 0.64 (1.01)	426.76 ± 20.79 (0.97)
	BC w/ Rational Trees	78.32 ± 0.86 (1.18)	464.41 ± 23.69 (1.06)
Fibonacci	AST	9.99 ± 0.20 (1.00)	10.49 ± 0.32 (1.00)
	Sub-Bytecodes	10.11 ± 0.05 (1.01)	11.91 ± 0.26 (1.14)
	Facts	10.47 ± 0.07 (1.05)	11.63 ± 0.35 (1.11)
	C-Interface	10.57 ± 0.07 (1.06)	3.86 ± 0.05 (0.37)
	AST w/ Rational Trees	10.16 ± 0.10 (1.02)	10.39 ± 0.27 (0.99)
	BC w/ Rational Trees	10.11 ± 0.06 (1.01)	10.72 ± 0.32 (1.02)
Fibonacci (Maxint)	AST	27.86 ± 0.52 (1.00)	191.46 ± 2.87 (1.00)
	Sub-Bytecodes	35.10 ± 0.31 (1.26)	231.86 ± 10.55 (1.21)
	Facts	41.42 ± 2.08 (1.49)	227.26 ± 9.98 (1.19)
	C-Interface	61.63 ± 1.45 (2.21)	32.19 ± 0.72 (0.17)
	AST w/ Rational Trees	28.62 ± 0.88 (1.03)	187.49 ± 7.31 (0.98)
	BC w/ Rational Trees	33.23 ± 1.01 (1.19)	200.34 ± 5.38 (1.05)
Generated	AST	16.53 ± 0.02 (1.00)	131.51 ± 3.86 (1.00)
	Sub-Bytecodes	24.89 ± 0.19 (1.51)	144.96 ± 2.86 (1.10)
	Facts	26.00 ± 0.93 (1.57)	140.42 ± 3.19 (1.07)
	C-Interface	NA	15.06 ± 0.29 (0.11)
	AST w/ Rational Trees	16.88 ± 0.04 (1.02)	129.03 ± 5.29 (0.98)
	BC w/ Rational Trees	20.41 ± 0.10 (1.23)	139.36 ± 6.72 (1.06)
Generated2	AST	24.31 ± 0.12 (1.00)	199.75 ± 5.85 (1.00)
	Sub-Bytecodes	37.26 ± 0.30 (1.53)	211.67 ± 5.09 (1.06)
	Facts	38.45 ± 1.36 (1.58)	204.97 ± 7.27 (1.03)
	C-Interface	NA	22.65 ± 0.72 (0.11)
	AST w/ Rational Trees	25.02 ± 0.06 (1.03)	193.39 ± 5.42 (0.97)
	BC w/ Rational Trees	30.20 ± 0.09 (1.24)	220.79 ± 7.53 (1.11)
Generated3	AST	15.98 ± 0.04 (1.00)	124.97 ± 3.00 (1.00)
	Sub-Bytecodes	24.14 ± 0.23 (1.51)	136.97 ± 3.92 (1.10)
	Facts	27.93 ± 0.95 (1.75)	133.31 ± 4.39 (1.07)
	C-Interface	NA	14.47 ± 0.43 (0.12)
	AST w/ Rational Trees	16.04 ± 0.03 (1.00)	121.68 ± 3.85 (0.97)
	BC w/ Rational Trees	19.43 ± 0.08 (1.22)	128.45 ± 3.90 (1.03)

Table 2: Mean runtimes in seconds including the 0.95 confidence interval. The value in parenthesis describes the normalised runtime (on the basis of the AST interpreter). Fastest runtimes per benchmark and interpreter are highlighted.

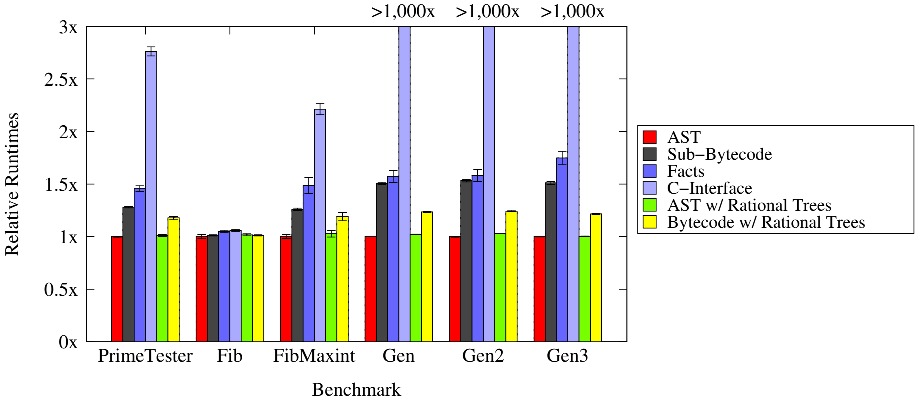


Fig. 18: Relative runtimes in SICStus, normalised to the runtime of the AST interpreter

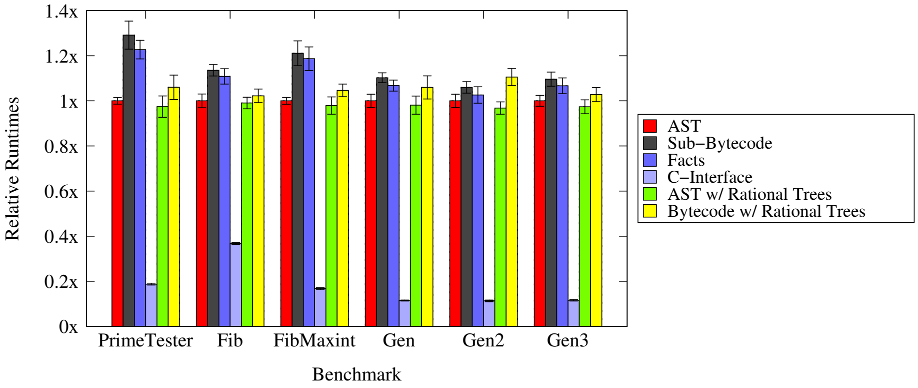


Fig. 19: Relative runtimes in SWI, normalised to the runtime of the AST interpreter

4.2 Results

The results of the benchmarks are shown in table 2. The mean value is determined by the geometric mean as proposed by [2]. For the interpreter based on SICStus' C-Interface, we cancelled the runs of the generated benchmarks after eight hours without a result.

The most important result is that using either Prolog implementation, the naive AST interpreter outperforms all of our pure Prolog implementations of bytecode interpreters.

Figure 18 shows the results specific for SICStus Prolog. Independent of the benchmark, all bytecode interpreters based on sub-bytecodes and on Prolog facts are slow in comparison. The AST interpreter utilising rational trees performs about as well as the naive AST interpreter. Surprisingly, the interpreter that dispatches in C suffers heavy performance issues. At the time of writing, we do not understand the reasons but are in contact with the SICStus support to clarify this behaviour.

The results utilising SWI Prolog are shown in fig. 19. In comparison to any bytecode-interpreter, the AST interpreter is faster. The AST interpreter with rational trees seems to be slightly more performant. However, the difference is small enough to be included in the uncertainty of measurement. The dispatching in C, however, is very fast. Depending on the benchmark, it can achieve a speed-up by an order of magnitude.

5 Conclusion, Related and Future Work

In this paper, we presented the language Acol and multiple ways to implement it as AST as well as bytecode interpreters. We designed several benchmarks in order to evaluate their performance using different implementations of Prolog.

Our results suggest that if an interpreter is to be implemented in Prolog, the implementation as an AST interpreter is very performant. Furthermore, it does not involve any compilation overhead as it can work on the data structure returned by the parser. Moreover, when using SWI Prolog, one can utilise C to efficiently implement the dispatching and query Prolog predicates for the domain logic.

In [6], Rossi and Sivalingam explored dispatching techniques in C based bytecode interpreters, with the result that a less portable approach of composing the code in memory before executing it yielded the best results. The techniques discussed in [6] could be used in combination with SWI to further improve the instruction dispatching performance in C.

An alternative for improving the execution time of a program, that was not discussed here, is partial evaluation [3]. We intend to investigate the impact of offline partial evaluation when compiling a subset of the described interpreters for our benchmarks.

However, Acol is a very simple language. Additional work is required to determine whether these findings are applicable for more complex languages. Furthermore, a richer language facilitates the creation of more benchmarks.

References

1. Manuel Carro. An Application of Rational Trees in a Logic Programming Interpreter for a Procedural Language. *CoRR*, cs.DS/0403028, March 2004.
2. Philip J. Fleming and John J. Wallace. How not to lie with statistics: The correct way to summarize benchmark results. *Commun. ACM*, 29(3):218–221, March 1986.
3. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.
4. Michael Leuschel and Michael J. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.
5. The PyPy Project. The Object Space, 2015.
6. M Rossi and K Sivalingam. A survey of instruction dispatch techniques for byte-code interpreters. *Seminar on Mobile Code*, 1996.

7. David H D Warren. An Abstract Prolog Instruction Set. Technical report, Artificial Intelligence Center - SRI International, 1983.

Time-predictable Computer Architecture to Simplify Worst-Case Execution Time Analysis

Martin Schoeberl

Technical University of Denmark

Abstract. Standard processors are optimized for performing best in the average case. Out-of-order pipelines, speculation, and several levels of caches are example features that improve average case execution time. However, for real-time systems we are interested in the worst-case execution time (WCET). And exactly those features that improve average case performance are problematic for the WCET. They may increase the WCET and they may be hard, or even impossible, to model for WCET analysis. In this paper we present a new line of computer architecture research where the main optimization point is for the WCET and especially to simplify WCET analysis.

E.V.A. - Electronic Visual Analysis: High-Performance Computing in a Heterogeneous Environment

Dietmar Schreiner

Vienna University of Technology

Abstract. Criminal investigators dealing with matters of child pornography are facing steadily increasing numbers of multimedia files to be examined. In Austria, the overall number of files per year has already exceeded 25 million. Seizure with several hundreds of thousand pictures and videos per case are no longer unusual but have become normal. All those files not only have to be forensically classified, but have to be linked to each other in order to convict serial perpetrators by extracting cross-case evidence. Project E.V.A. aims at developing a computer aided methodology that simplifies the actual work of criminal investigators and provides support for complex cross-case investigations. Taking the huge actual number of media files under investigation into account, it becomes obvious that computing time and energy consumption will become a serious bottleneck for a computer aided investigative methodology. Hence, the key research issues of our project not only cope with algorithms of computer vision and machine learning but also with their high performance implementation on various computing platforms like multi-core CPUs, GPUs, and FPGAs.

Transactional Tries*

Michael Schröder

Vienna University of Technology
mc.schroeder@gmail.com
http://github.com/mcschroeder

Abstract Software Transactional Memory (STM) immensely simplifies concurrent programming by allowing memory operations to be grouped together into atomic blocks. But a common problem with STM is contention. Many standard data structures, when used in a transactional setting, cause unreasonably high numbers of conflicts. I present a contention-free STM data structure for Haskell: the transactional trie. It is based on the lock-free concurrent trie, but lifted into an STM context. It uses well-considered local side-effects to eliminate unnecessary conflicts while preserving transactional safety.

1 Introduction

It is a widely held opinion that concurrent programming is difficult and error-prone. Low-level synchronization mechanisms, such as locks, are notoriously tricky to get right. Deadlocks, livelocks, heisenbugs and other issues encountered when writing complex concurrent systems are usually hard to track down and often confound even experienced programmers.

To simplify concurrent programming, higher-level abstractions are needed. One such abstraction is *Software Transactional Memory* (STM). Briefly, this technique allows the programmer to group multiple memory operations into a single atomic block, not unlike a database transaction. When implemented in a high-level language such as Haskell, with its emphasis on purity and its strong static type system, STM becomes especially powerful.

The fundamental data type of STM is the transactional variable. A TVar stores arbitrary data, to be accessed and modified in a thread-safe manner. For example, I might define a bank account as

```
type Euro = Int
type Account = TVar Euro
```

and then use a function like

```
transfer :: Account → Account → Euro → STM ()
```

to safely — in the transactional sense — move money between accounts.

* This report is an abridged version of chapter 3 of my master's thesis [10].

But where do those accounts come from? If I am a bank, how do I represent the whole collection of accounts I manage, in a way that is transactionally safe? The obvious solution, and a common pattern, is to simply use an existing container type and put that type into a TVar:

```
type IBAN = String
type Bank = TVar (Map IBAN Account)
```

Since looking up an account from the Map involves a readTVar operation, the Map is entangled with the transaction, and I can be sure that when transferring money between accounts, both accounts actually exist in the bank at the time when the transaction commits.

The drawback of this pattern of simply wrapping a Map inside a TVar is that when adding or removing elements of the Map, one has to replace the Map inside the TVar wholesale. Thus all concurrently running transactions that have accessed the Map become invalid and will have to restart once they try to commit. Depending on the exact access patterns, this can be a serious cause of contention. For example, one benchmark running on a 16-core machine, with 16 threads each trying to commit a slice out of 200 000 randomly generated transactions, resulted in over 1.3 million retries. That is some serious overhead!

The underlying problem is that the whole Map is made transactional, when we only ever care about the subset of the Map that is relevant to the current transaction. If transaction A updates an element with key k_1 and transaction B deletes an element with key k_2, then those two transactions only conflict if $k_1 = k_2$; if k_1 and k_2 are different, then there is no reason for either of the transactions to wait for the other one. But the Map does not know it is part of a transaction, and the TVar does not know nor care about the structure of its contents. And so the transactional net is cast too wide.

The solution is to not simply put a Map, or any other ready-made container type, into a TVar, but to design data structures specifically tailored to the needs of transactional concurrency. In this report, I present one such data structure: the *transactional trie*.

2 Background: STM in Haskell

Here are the main data types and operations of STM in Haskell:

```
data STM a
instance Monad STM

atomically :: STM a → IO a

data TVar a
newTVar :: a → STM (TVar a)
readTVar :: TVar a → STM a
writeTVar :: TVar a → a → STM ()

retry :: STM a
orElse :: STM a → STM a → STM a
```

Atomic blocks in Haskell are represented by the STM monad. Inside this monad, we can freely operate on transactional variables, or TVars. We can read them, write them and create new ones. When we want to actually perform an STM computation and make its effects visible to the rest of the world, we apply atomically to the computation. This function turns an STM block into a transaction in the IO monad that, when executed, will take place atomically with respect to all other transactions.

For example, the following code snippet increments a transactional variable named v:

```
atomically $ do x ← readTVar v
                writeTVar v (x + 1)
```

The use of atomically guarantees that no other thread can come in between the reading and writing of the variable. The sequence of operations happens indivisibly.

An important aspect of Haskell's STM implementation is that it is fully composable. Smaller transactions can be combined into larger transactions without having to know how these smaller transactions are implemented. An important tool to make this possible is the composable blocking operator retry. Conceptually, retry abandons the current transaction and runs it again from the top. In the following example, the variable v is decremented, unless it is zero, in which case the transaction blocks until v is non-zero again:

```
atomically $ do x ← readTVar v
                if x ≡ 0
                  then retry
                  else writeTVar v (x − 1)
```

In addition to retry, there is the orElse combinator, which allows "trying out" transactions in sequence. m1 'orElse' m2 first executes m1; if m1 returns, then orElse returns; but if m1 retries, its effects are discarded and m2 is executed instead.

STM is also robust against exceptions. The standard functions throw and catch act as expected: if an exception occurs inside an atomic block and is not caught, the transaction's effects are discarded and the exception is propagated.

For more background on Haskell's STM, including its implementation, see the original STM papers [3, 2]. For a more thorough exploration of not only STM but also other Haskell concurrency mechanisms, read Simon Marlow's excellent book on that topic [5].

3 Transactional Tries

The transactional trie is based on the concurrent trie of Prokopec, Bagwell, and Odersky [8], which is a non-blocking concurrent version of the hash array mapped trie first described by Bagwell [1].

A hash array mapped trie is a tree whose leaves store key-value bindings and whose nodes are implemented as arrays. Each array has 2^k elements. To look up a key, you take the initial k bits of the key's hash as an index into the root array. If the element at that index is another array node, you continue by using the next k bits of the hash as an index into that second array. If that element is another array, you again use the next k bits of the hash, and so on. Generally speaking, to index into an array node at level l, you use the k bits of the hash beginning at position $k * l$. This procedure is repeated until either a leaf node is found or one of the array nodes does not have an entry at the particular index, in which case the key is not yet present in the trie. The expected depth of the trie is $O(log_{2^k}(n))$, which means operations have a nice worst-case logarithmic performance.

Most of the array nodes would only be sparsely populated. To not waste space, the arrays are actually used in conjunction with a bitmap of length 2^k that encodes which positions in the array are actually filled. If a bit is set in the bitmap, then the (logical) array contains an element at the corresponding index. The actual array only has a size equal to the bit count of the bitmap, and after obtaining a (logical) array index i in the manner described above, it has to be converted to an index into the sparse array via the formula $\#((i-1) \wedge bmp)$, where $\#$ is a function that counts the number of bits and bmp is the array's bitmap. To ensure that the bitmap can be efficiently represented, k is usually chosen so that 2^k equals the size of the native machine word, e.g. on 64-bit systems $k = 6$.

The *concurrent* trie extends the hash trie by adding *indirection nodes* above every array node. An indirection node simply points to the array node underneath it. Indirection nodes have the property that they stay in the trie even if the nodes above or below them change. When inserting an element into the trie, instead of directly modifying an array node, an updated copy of the array node is created and an atomic compare-and-swap operation on the indirection node is used to switch out the old array node for the new one. If the compare-and-swap operation fails, meaning another thread has already modified the array while we were not looking, the operation is retried from the beginning. This simple scheme, where indirection nodes act as barriers for concurrent modification, ensures that there are no lost updates or race conditions of any kind, while keeping all operations completely lock-free. A more thorough discussion, including proofs of linearizability and lock-freedom, can be found in the paper by Prokopec et al. [8]. A Haskell implementation of the concurrent trie, as a mutable data structure in IO, is also available [9].

The *transactional* trie is an attempt to lift the concurrent trie into an STM context. The idea is to use the lock-freedom of the concurrent trie to make a non-contentious data structure for STM. This is not entirely straightforward, as there is a natural tension between the atomic compare-and-swap operations of the concurrent trie, which are pessimistic and require execution inside the IO monad, and optimistic transactions as implemented by STM. While it is possible to simulate compare-and-swap using TVars and retry, this would entangle the indirection nodes with the rest of the transaction, which is exactly the opposite of what we want. To keep the non-blocking nature of the concurrent trie, the indirection nodes need to be kept independent of the transaction as a whole, which should only hinge on the actual values stored in the trie's leaves. If two transactions were to cross paths at some indirection node, but otherwise concern independent elements of the trie, then neither transaction should have to retry or block. Side-effecting compare-and-swap operations that run within but independently of a transaction are the only way to achieve this. Alas, the type system, with good reason, will not just allow us to mix IO and STM actions, so we have to circumvent it from time to time using unsafeIOToSTM. We will need to justify every single use of unsafeIOToSTM and ensure it does not lead to violations of correctness. Still, bypassing the type system is usually a bad sign, and indeed we will see that correctness can only be preserved at the cost of memory efficiency, at least in an STM implementation without finalizers.

4 Implementation

The version of the transactional trie discussed in this report is available on Hackage at `http://hackage.haskell.org/package/ttrie-0.1.2`. The full source code can also be found at `http://github.com/mcschroeder/ttrie`.

The module Control.Concurrent.STM.Map[1] exports the transactional trie under the following interface:

```
data Map k v
empty :: STM (Map k v)
insert :: (Eq k, Hashable k) ⇒ k → v → Map k v → STM ()
lookup :: (Eq k, Hashable k) ⇒ k → Map k v → STM (Maybe v)
delete :: (Eq k, Hashable k) ⇒ k → Map k v → STM ()
```

Now let us implement it. As always, we begin with some types:[2]

```
newtype Map k v = Map (INode k v)
type INode k v = IORef (Node k v)
data Node k v = Array !(SparseArray (Branch k v))
              | List  ![Leaf k v]
data Branch k v = I !(INode k v)
                | L !(Leaf k v)
data Leaf k v = Leaf !k !(TVar (Maybe v))
```

The transactional trie largely follows the construction of a concurrent trie:

[1] The name of the trie's public data type is Map, instead of, say, TTrie. The more general name is in keeping with other container libraries and serves to decouple the interface from the specific implementation based on concurrent tries.

[2] The ! operator is a strictness annotation.

- The INode is the indirection node described in the previous section and is simply an IORef, which is a mutable variable in IO. To read and write IORefs atomically, we will use some functions and types from the `atomic-primops` package [6]:

  ```
  data Ticket a
  readForCAS :: IORef a → Ticket a
  peekTicket :: Ticket a → IO a
  casIORef :: IORef a → Ticket a → a → IO (Bool, Ticket a)
  ```

  The idea of the Ticket type is to encapsulate proof that a thread has observed a specific value of an IORef. Due to compiler optimizations, it would not be safe to just use pointer equality to compare values directly.

- A Node is either an Array of Branches or a List of Leafs. The List is used in case of hash collisions. A couple of convenience functions help us manipulate such collision lists:

  ```
  listLookup :: Eq k ⇒ k → [Leaf k v] → Maybe (TVar (Maybe v))
  listDelete :: Eq k ⇒ k → [Leaf k v] → [Leaf k v]
  ```

  Their implementations are entirely standard.
  The Array is actually a SparseArray, which abstracts away all the bit-fiddling necessary for navigating the bit-mapped arrays underlying a hash array mapped trie. Its interface is largely self-explanatory:

  ```
  data SparseArray a
  emptyArray :: SparseArray a
  mkSingleton :: Level → Hash → a → SparseArray a
  mkPair :: Level → Hash → a → a → Maybe (SparseArray a)
  arrayLookup :: Level → Hash → SparseArray a → Maybe a
  arrayInsert :: Level → Hash → a → SparseArray a → SparseArray a
  arrayUpdate :: Level → Hash → a → SparseArray a → SparseArray a
  ```

  I will not go into the implementation of SparseArray. It is fairly low-level and can be found in the internal Data.SparseArray module of the `ttrie` package.
  Some additional functions are used to manipulate Hashes and Levels. Again, they are self-explanatory:

  ```
  type Hash = Word
  hash :: Hashable a ⇒ a → Hash
  type Level = Int
  down :: Level → Level
  up :: Level → Level
  lastLevel :: Level
  ```

  - A Branch either adds another level to the trie by being an INode or it is simply a single Leaf.

The one big difference to a concurrent trie lies in the definition of the Leaf. Basically, a Leaf is a key-value mapping. It stores a key k and a value v. But the way it stores v determines how the trie behaves in a transactional context. Let us build it step by step:

1. Imagine if Leaf were defined exactly like in a concurrent trie:

   ```
   data Leaf k v = Leaf !k v
   ```

   Then an atomic compare-and-swap on an INode to insert a new Leaf would obviously not be safe during an STM transaction: other transactions could see the new value v before our transaction commits; and they could replace v by inserting a new Leaf for the same key, resulting in our insert being lost.

2. We can eliminate lost inserts by wrapping the value in a TVar:

   ```
   data Leaf k v = Leaf !k !(TVar v)
   ```

   Now, instead of replacing the whole Leaf to update v, we can use writeTVar to only modify the value part of the Leaf. If two transactions try to update the same Leaf, then STM will detect the conflict and one of the transactions would have to retry.
   Of course, if there does not yet exist a Leaf for a specific key, then a new Leaf will still have to be inserted with a compare-and-swap. In this case it is again possible for other transactions to read the TVar immediately after the swap, even though our transaction has not yet committed and may still abort. This can happen without conflict because the new Leaf contains a newly allocated TVar and allocation effects are allowed to escape transactions by design. Reading a newly allocated TVar will never cause a conflict.

3. To ensure proper isolation, the actual type of Leaf looks like this:

   ```
   data Leaf k v = Leaf !k !(TVar (Maybe v))
   ```

   By adding the Maybe, we can allocate new TVars with Nothing in them. A transaction can then insert a new Leaf containing Nothing using the compare-and-swap operation. Other threads will still able to read the new TVar immediately after the compare-and-swap, but all they will get is Nothing. The transaction, meanwhile, can simply writeTVar (Just v) to safely insert the actual value into the Leaf's TVar. If another transaction also writes to the TVar and commits before us, then we have a legitimate conflict on the value level, and our transaction will simply retry.

Now that we have the types that make up the trie's internal structure, we can implement its operations. We begin with the function to create an empty trie:

```
empty :: STM (Map k v)
empty = unsafeIOToSTM $ Map <$> newIORef (Array emptyArray)
```

It contains no surprises, although it has the first use of unsafeIOToSTM, which in this case is clearly harmless.

For the rest of the operations, let us assume we have a function

```
getTVar :: (Eq k, Hashable k) ⇒ k → Map k v → STM (TVar (Maybe v))
```

that either returns the TVar stored in the Leaf for a given key, or allocates a new TVar for that key and inserts it appropriately into the trie. The TVar returned by getTVar k m will always either contain Just v, where v is the value associated with the key k in the trie m, or Nothing, if k is not actually present in m. Additionally, getTVar obeys the following invariants:

Invariant 1: $\mathsf{getTVar}\ k_1\ m \equiv \mathsf{getTVar}\ k_2\ m \iff k_1 \equiv k_2$
Invariant 2: getTVar itself does not read from nor write to any TVars.

Now we can define the trie's operations as follows:

```
insert k v m = do var ← getTVar k m
                  writeTVar var (Just v)
lookup k m = do var ← getTVar k m
                readTVar var
delete k m = do var ← getTVar k m
                writeTVar var Nothing
```

The nice thing about defining the operations this way, is that correctness and non-contentiousness follow directly from the invariants of getTVar. The first invariant ensures correctness. If we get the same TVar every time we call getTVar with the same key, and if that TVar is unique to that key, then STM will take care of the rest. And if, by the second invariant, getTVar does not touch any transactional variables, then the only way one of the operations can cause a conflict is if it actually operates at the same time on the same TVar as another transaction. Unnecessary contention is therefore not possible.

All that is left to do is implementing getTVar. Essentially, getTVar is a combination of the insert and lookup functions of the concurrent trie, just lifted into STM. It tries to look up the TVar associated with a given key, and if that does not exist, allocates and inserts a new TVar for that key. When inserting a new TVar, the structure of the trie has to be changed to accommodate the new element.

Let us look at the code:

```
getTVar k (Map root) = go root 0
  where
    h = hash k
```

The actual work is done by the recursive helper function go. It begins at level 0 by looking into the *root* indirection node. Note that throughout the iterations of go, the hash h of the key is only computed once.

```
go inode level = do
  ticket ← unsafeIOToSTM $ readForCAS inode
  case peekTicket ticket of
    Array a → case arrayLookup level h a of
      Just (I inode2) → go inode2 (down level)
      Just (L leaf2@(Leaf k2 var))
         | k ≡ k2      → return var
         | otherwise  → cas inode ticket (growTrie level a (hash k2) leaf2)
      Nothing         → cas inode ticket (insertLeaf level a)
    List xs  → case listLookup k xs of
      Just var        → return var
      Nothing         → cas inode ticket (return ∘ List ∘ (:xs))
```

The use of unsafeIOToSTM here is clearly safe — all we are doing is reading the value of the indirection node. This does not have any side effects, so it does not matter if the transaction aborts prematurely. If the transaction retries, the indirection node is just read again — possibly resulting in a different value. It is also possible that the value of the indirection node changes during the runtime of the rest of the function — but that is precisely why we obtain a Ticket.

Depending on the contents of the indirection node, we either go deeper into the trie with a recursive call of go; return the TVar associated with the key; or insert a new TVar by using the cas function to swap out the old contents of the indirection node with an updated version that somehow contains the new TVar.

The cas function is also part of the **where** clause of getTVar:

```
cas inode ticket f = do
  var ← newTVar Nothing
  node ← f (Leaf k var)
  (ok, _) ← unsafeIOToSTM $ casIORef inode ticket node
  if ok then return var
       else go root 0
```

It implements a transactionally safe compare-and-swap procedure:

1. Allocate a new TVar containing Nothing.
2. Use the given function f to produce a *node* containing a Leaf with this TVar.
3. Use casIORef to compare-and-swap the old contents of the *inode* with the new *node*.
4. If the compare-and-swap was successful, the new *node* is immediately visible to all other threads. Return the TVar to the caller, who is now free to use writeTVar to fill in the final value.
5. If the compare-and-swap failed, because some other thread has changed the *inode* since the time we first read it, restart the operation — not with the STM retry, which would restart the whole transaction, but simply by calling go *root* 0 again.

All that is remaining now are the functions given for f in the code of go. Given a new Leaf, they are supposed to return a Node that somehow contains this new Leaf. In the case of the overflow list, this is just a trivial anonymous function that prepends the leaf into the List node. The insertLeaf function does pretty much the same, except for Array nodes:

```
insertLeaf level a leaf = do
  let a′ = arrayInsert level h (L leaf) a
  return (Array a′)
```

In case of a key collision, things are a slightly more involved. The growTrie function puts the colliding leaves into a new level of the trie, where they hopefully will not collide anymore:

```
growTrie level a h2 leaf2 leaf1 = do
  inode2 ← unsafeIOToSTM $ combineLeaves (down level) h leaf1 h2 leaf2
  let a′ = arrayUpdate level h (I inode2) a
  return (Array a′)
combineLeaves level h1 leaf1 h2 leaf2
   | level ⩾ lastLevel = newIORef (List [leaf1, leaf2])
   | otherwise =
     case mkPair level h (L leaf1) h2 (L leaf2) of
       Just pair → newIORef (Array pair)
       Nothing → do
         inode ← combineLeaves (down level) h1 leaf1 h2 leaf2
         let a = mkSingleton level h (I inode)
         newIORef (Array a)
```

The use of casIORef here is once again harmless, as combineLeaves only uses IO to allocate new IORefs. The mkPair function for making a two-element SparseArray returns a Maybe, because it is possible that on a given level of the trie the two keys hash to the same array index and so the leaves cannot both be put into a single array. In that case, another new indirection node has to be introduced into the trie and the procedure repeated. If at some point the last level has been reached, the leaves just go into an overflow List node.

5 Memory efficiency

While the transactional trie successfully carries over the lock-freedom of the concurrent trie and keeps the asymptotic performance of its operations, it does have to make a couple of concessions regarding memory efficiency.

The first concession is that when looking up any key for the first time, the lookup operation will actually grow the trie. This is a direct consequence of using getTVar to implement the trie's basic operations. If getTVar does not find the Leaf for a given key, it allocates a new one and inserts it. One might wonder if it is possible to implement a lookup function that does not rely on getTVar. The following attempt is pretty straightforward and appears to be correct at first glance — although you might already guess from its name that something is not quite right:

```
phantomLookup :: (Eq k, Hashable k) ⇒ k → Map k v → STM (Maybe v)
phantomLookup k (Map root) = go root 0
  where
    h = hash k

    go inode level = do
      node ← unsafeIOToSTM $ readIORef inode
      case node of
        Array a → case arrayLookup level h a of
          Just (I inode₂) → go inode₂ (down level)
          Just (L (Leaf k₂ var))
             | k ≡ k₂      → readTVar var
             | otherwise  → return Nothing
          Nothing         → return Nothing
        List xs  → case listLookup k xs of
          Just var        → readTVar var
          Nothing         → return Nothing
```

The problem with this simple implementation is that under certain circumstances it allows for *phantom reads*. Consider the following pair of functions:

```
f = atomically $ do v1 ← phantomLookup k
                    v2 ← phantomLookup k
                    return (v1 ≡ v2)
g = atomically (insert k 23)
```

Due to STM's isolation guarantees, one would reasonably expect that f always returns True. However, sometimes f will return False when g is run between the two phantomLookups in f. How is this possible? If you start out with an empty trie, then the first phantomLookup in f obviously returns Nothing. And it does so without touching any TVars, because there is no TVar for k at this point. Only when running g for the first time, will a TVar for k be created. The transaction inside f will now happily read from this TVar during the second phantomLookup and will not detect any inconsistencies, because this is the first time it has seen the TVar. This problem does not only occur on an empty trie, but any time we look up a key that has not previously been inserted. The only remedy is to ensure that there is always a TVar for every key, even if it is filled with Nothing, which is exactly what the implementation of lookup using getTVar does.

Granted, it seems as if these kinds of phantom lookups might not occur regularly in practice, and even if they did, they would probably cause no great harm. The overhead of always allocating a Leaf for every key that is ever looked up, on the other hand, seems much more troublesome. However, phantomLookup exhibits exactly the kind of seldom-occurring unexpected behavior that results in bugs that are incredibly hard to find. And having a lookup function that grows the trie is really only an issue in two cases:

1. when we expect the keys we look up to not be present a significant amount of the time; then a transactional trie is probably really not the right data structure. Although if one were to use phantomLookup instead of lookup, and if in this particular scenario phantom lookups are actually acceptable, then using a transactional trie could still be feasible.
2. when a malicious actor purposefully wants to increase memory consumption, i.e. a classic denial-of-service attack; then one can again counteract this by using phantomLookup, limited to those places that are susceptible to attack. For example, a login routine in a web application could first use phantomLookup to check if the user actually exists, before continuing with the transaction. Here the phantom lookup does not matter, because if the user does not exist the transaction is aborted anyway.

Thus, it makes sense to have the behavior of lookup be the default and provide phantomLookup for those select scenarios where it is actually an improvement.

The other trade-off the trie has to make regarding memory efficiency, is that the delete operation does not actually remove Leafs or compact the trie again. It merely fills a Leaf's TVar with Nothing. This frees up the values associated with the keys, which is the major part of a trie's memory consumption, but it does not delete the keys or compress the structure that has emerged in the trie, which might now be suboptimal given the trie's current utilization.

Again, for the common use case, this might not be a problem. Very often, we do not want to actually delete certain data, but merely mark it as deleted; or maybe delete the data, but mark the associated keys as having been previously in use in order to prevent reusing them. Think of unique user IDs, for example. In such a scenario, the overhead of the trie not actually deleting Leafs disappears.[3]

6 Evaluation

I empirically evaluated the transactional trie against similar data structures, measuring contention, runtime performance and memory allocation. The benchmarks were run on an Amazon EC2 C3 extra-large instance with Intel Xeon E5-2680 v2 (Ivy Bridge) processors and a total of 16 physical cores. Under comparison were three hashing-based container types: a transactional trie; a HashMap from the `unordered-containers` library [11], wrapped inside a TVar; and the STM-specialized hash array mapped trie from the `stm-containers` library [13].

The same random Text strings are used as keys for each container. Each benchmark consists of a number of random STM transaction. The transactions

[3] For the cases where we do want the trie to always be as compact a representation of its data as possible, there is an unsafeDelete operation, which really does remove Leafs and compresses the trie again. Alas, as its name suggests, unsafeDelete is not transactionally safe. It can be made safe by using an STM extension called *finalizers*. For a thorough description of finalizers and how they can be used to make unsafeDelete safe, see [10].

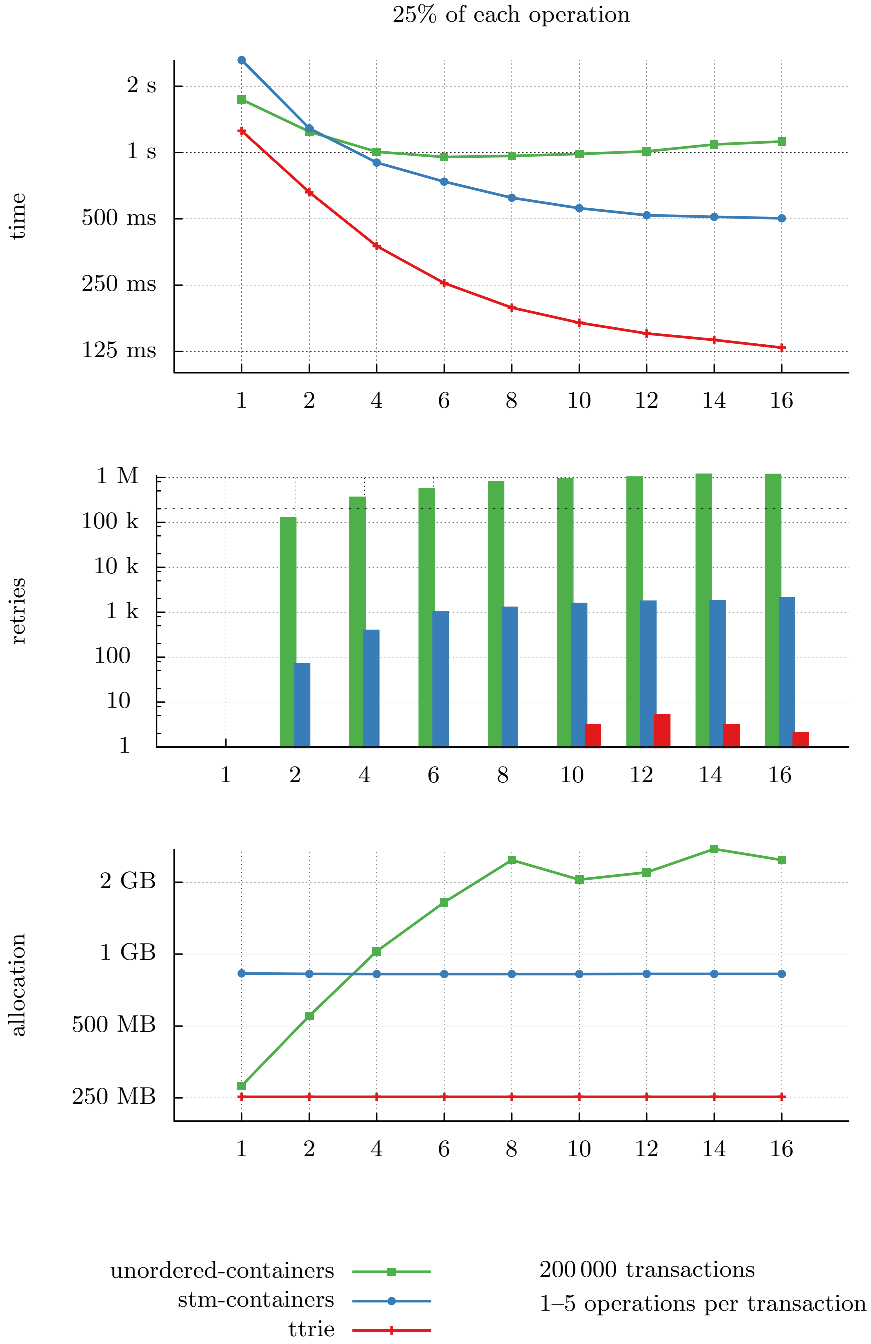


Figure 1. Benchmark comparing STM data structures

are split evenly over the number of threads in use. The time it takes to complete all transactions for a particular container is measured using the criterion and criterion-plus libraries [7, 12], which calculate the mean execution time over many iterations. To measure contention, the transactions are run again using the stm-stats library [4] to count how often the STM runtime system has to restart transactions due to conflicts. Finally, the transactions are run once more to measure the total amount of allocated memory, using GHCs built-in facilities for collecting memory usage statistics. The benchmark was compiled using GHC 7.8.3. For more details about test data generation and the exact benchmark setup, see the ttrie source distribution.

Figure 1 shows the results of a benchmark performing 200 000 random STM transactions, where each transaction performs 1–5 operations. Each operation (insert, lookup, update or delete) occurs equally likely in the mix of operations per transaction and the containers are prefilled with 1 000 000 entries.[4]

As we can see from the number of retries, the transactional trie exhibits no contention; the handful of retries it has to perform — 5 in the worst case — are due to legitimate conflicts. This is in stark contrast to unordered-containers and stm-containers: here, the number of spurious retries vastly overshadows the legitimate conflicts. In the worst case, the TVar-wrapped HashMap has to retry more than 1.3 million times for the 200 000 transactions to succeed. The run-time performance of unordered-containers begins to rapidly degrade at 4 threads, which is when the number of retries first exceeds the number of transactions.

Overall, ttrie is 2–4 times faster than stm-containers, allocating only a third of the memory; and 1.3–8.6 times faster than unordered-containers, allocating almost 10 times less memory.

References

[1] Phil Bagwell. *Ideal Hash Trees.* Tech. rep. LAMP-REPORT-2001-001. EPFL, 2001.

[2] Tim Harris and Simon Peyton Jones. “Transactional memory with data invariants.” In: *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing.* TRANSACT ’06. Ottawa, Canada: ACM, 2006.

[3] Tim Harris et al. “Composable Memory Transactions.” In: *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* PPoPP ’05. Chicago, IL, USA: ACM, 2005, pp. 48–60.

[4] David Leuschner, Stefan Wehr, and Joachim Breitner. *stm-stats: retry statistics for STM transactions.* Version 0.2.0.0. 2011. URL: http://hackage.haskell.org/package/stm-stats-0.2.0.0.

[5] Simon Marlow. *Parallel and Concurrent Programming in Haskell.* O’Reilly Media Inc., 2013.

[4] For additional benchmarks with different setups, see [10].

[6] Ryan Newton. *`atomic-primops`: A safe approach to CAS and other atomic ops in Haskell.* Version 0.6. 2014. URL: `http://hackage.haskell.org/package/atomic-primops-0.6`.

[7] Bryan O'Sullivan. *`criterion`: Robust, reliable performance measurement and analysis.* Version 1.0.2.0. 2014. URL: `http://hackage.haskell.org/package/criterion-1.0.2.0`.

[8] Aleksander Prokopec, Phil Bagwell, and Martin Odersky. *Cache-Aware Lock-Free Concurrent Hash Tries.* Tech. rep. EPFL-REPORT-166908. EPFL, 2011.

[9] Michael Schröder. *`ctrie`: Non-blocking concurrent map.* Version 0.1.0.2. 2014. URL: `http://hackage.haskell.org/package/ctrie-0.1.0.2`.

[10] Michael Schröder. "Durability and Contention in Software Transactional Memory." MSc Thesis. Vienna University of Technology, 2015. URL: `http://github.com/mcschroeder/thesis`.

[11] Johan Tibell and Edward Z. Yang. *`unordered-containers`: Efficient hashing-based container types.* Version 0.2.5.0. 2014. URL: `http://hackage.haskell.org/package/unordered-containers-0.2.5.0`.

[12] Nikita Volkov. *`criterion-plus`: Enhancement of the `criterion` benchmarking library.* Version 0.1.3. 2014. URL: `http://hackage.haskell.org/package/criterion-plus-0.1.3`.

[13] Nikita Volkov. *`stm-containers`: Containers for STM.* Version 0.2.3. 2014. URL: `http://hackage.haskell.org/package/stm-containers-0.2.3`.

A Data-Flow Approach for Compiling the Sequentially Constructive Language (SCL)

Steven Smyth, Christian Motika, and Reinhard von Hanxleden

Real-Time and Embedded Systems Group, Department of Computer Science
Christian-Albrechts-Universität zu Kiel, Olshausenstr. 40, 24118 Kiel, Germany
www.informatik.uni-kiel.de/rtsys/
{ssm,cmot,rvh}@informatik.uni-kiel.de

Abstract. The Sequentially Constructive Language (SCL) is a minimal synchronous language that captures the essence of the Sequentially Constructive Model of Computation (SCMoC), a recently proposed extension of the classical synchronous model of computation. The SCMoC uses sequential scheduling information to increase the class of constructive (legal) synchronous programs. This facilitates the adoption of synchronous programming by users familiar with sequential programming in C or Java, thus simplifying the design of concurrent reactive/embedded systems with deterministic behavior. The theoretical foundations of the SCMoC are fairly well covered by now, and also the upstream compilation from SCCharts (a Statechart dialect) and SCEst (a variant of Esterel) to SCL. In this paper, we focus on how to compile SCL down to data-flow equations, which ultimately can be synthesized to hardware or executed in software.

1 Motivation & Related Work

Reactive systems are characterized by their regular interaction with the environment, typically under real-time constraints. Physical time is conceptually divided into a sequence of discrete *ticks*, and during each tick, the reactive system reads inputs from the environment, processes them according to some internal system state, and then updates the system state and produces outputs to the environment. Reactive systems may implement safety-critical applications where determinate behavior is essential. However, they often entail concurrent threads of control and interact through shared memory, which makes determinacy challenging. This has motivated the development of *synchronous languages* [3], which have been used successfully in the industry since the 1990s, e. g., for the development of avionics software or power plant control. Edwards [4] and Potop-Butucaru et al. [11] provide good overviews of compilation challenges and approaches for concurrent languages, including synchronous languages.

Synchronous languages achieve determinacy for concurrent systems by the *synchrony hypothesis*, which abstracts from computation time and thus assumes outputs to occur synchronously with the inputs they react to. This is achieved by demanding unique, stable values for all shared variables throughout each tick.

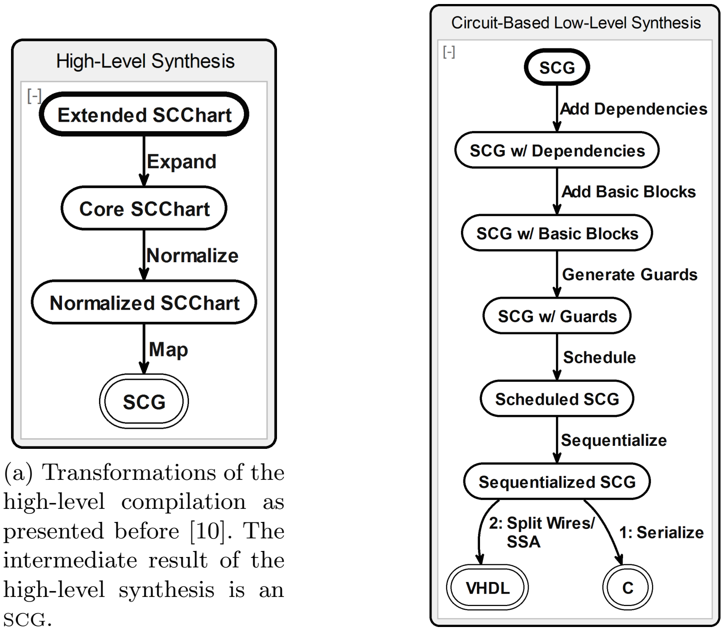
This simplifies formal reasoning, and has a natural, physical analogy of well-defined, stable voltages on all wires on a hardware circuit. We thus say that a synchronous program is *causal*, or *constructive*, if and only if it corresponds to a circuit where all wires have well-defined voltages for all possible inputs and all possible internal states, independent of variations in signal propagation delays in an actual hardware realization.

This Synchronous Model of Computation (SMoC) for concurrent programming contrasts with, e. g., the programming model offered by Java or Posix threads. There the outcome of a program with concurrent threads that share variables may depend on run-time scheduling decisions out of control of the programmer. In Java, achieving determinacy often requires additional, brittle constructs such as semaphores, monitors, barrier synchronizations etc. [9].

However, the classic realization of the synchrony hypothesis comes with restrictions that may be difficult to realize, in particular for novice programmers used to imperative languages such as C or Java. Specifically, the requirement that shared variables cannot change within a tick may come as a surprise. For example, a construct such as "if (x) { x = false }" would be forbidden in the classical SMoC, because x could be true and false within a tick. Instead, one could write for example "if (pre(x)) { x = false }" which states that if x was true in the *previous* tick, then set it to false in the current tick. This, however, would introduce an artificial tick boundary in a computation that conceptually has nothing to do with the passage of physical time. Conversely, the computation in question is purely sequential, with an obvious order of computation: first the test whether x is true, second the possible assignment to false. Thus there is no race condition between the read and the write of x, and a compiler should have no difficulty to produce determinate code.

The desire to combine the foundations and nice properties of synchronous languages with instantaneous memory updates has motivated the development of the *Sequentially Constructive* model of computation (SCMoC) [17,1]. The basic idea is to use any sequential scheduling information in the program to schedule computations in a determinate fashion, and to use a particular scheduling protocol to order concurrent variable accesses. The SCMoC may still reject certain programs as not being sequentially constructive, such as "fork x=y par y=x join", where the SCMoC does not know how to order the concurrent accesses to x and y and which is therefore not determinate. However, the SCMoC accepts many more programs than the SMoC.

The SCMoC is formally defined with the *Sequentially Constructive Graph* (SCG), which is textually represented as the *Sequentially Constructive Language* (SCL). The SCL is rather low-level and very simple yet rich enough to be used as an intermediate language for compiling higher-level languages that want to build on the SCMoC. So far, two such higher-level languages have been proposed. The first language is *Sequentially Constructive Statecharts* (*SCCharts*) [15], a dialect of Harel's statecharts [7]. The second language is *Sequentially Constructive Esterel* (SCEst) [12], an extension of Esterel [11].



(a) Transformations of the high-level compilation as presented before [10]. The intermediate result of the high-level synthesis is an SCG.

(b) Transformations of the low-level data-flow synthesis.

Fig. 1. Single-Pass Language-Driven Incremental Compilation (SLIC) approach transforming SCCharts to code.

Outline and Contributions

The original paper on SCCharts [15] briefly sketched two approaches to compile SCL further into software or hardware, namely the *data-flow approach* and the *priority-based approach*. We here present the data-flow approach in detail.

In Sec. 2 we explain the SCL and recapitulate definitions that are important for the remainder of the paper. The previously presented compilation chain (cf. Fig. 1) shows that the data-flow approach transforms from the generated SCG to a sequentialized variant and then eventually into code. This transformation is executed in several steps depicted in Fig. 1b. Each step is described in Sec. 3, which is the technical core of this paper. We here follow the Single-Pass Language-Driven Incremental Compilation (SLIC) approach [10] where every step is executed as a Model-to-Model (M2M) transformation, which facilitates a modular compilation chain.

We have validated our compilation chain with a range of test cases. This includes fairly extensive use in the class room. In Sec. 4 we report on a medium-sized railway project that uses the data-flow approach to synthesize a railway controller. We conclude in Sec. 5.

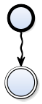
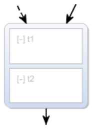
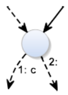
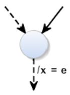
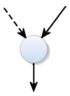
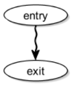
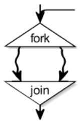
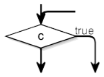
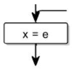
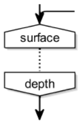
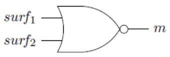
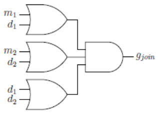
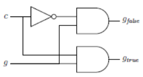
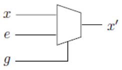
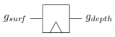
	Region (Thread)	**Superstate (Concurrency)**	**Trigger (Conditional)**	**Effect (Assignment)**	**State (Delay)**
SCCharts					
SCG					
SCL	t	fork t_1 par t_2 join	if (c) s_1 else s_2	$x = e$	pause
Data-Flow Code	$d = g_{exit}$ $m = \neg \bigvee_{surf \in t} g_{surf}$	$g_{join} = (d_1 \vee m_1) \wedge (d_2 \vee m_2) \wedge (d_1 \vee d_2)$	$g = \bigvee g_{in}$ $g_{true} = g \wedge c$ $g_{false} = g \wedge \neg c$	$g = \bigvee g_{in}$ $x' = g \; ? \; e : x$	$g_{depth} =$ pre(g_{surf})
Circuits		 m_1 d_1 m_2 d_2 d_1 d_2 g_{join}			

Fig. 2. Matrix showing the entire mapping throughout the transformation process from SCCharts to circuits.

2 The Sequentially Constructive Language (SCL)

This section gives an overview of SCL. It only gives the necessary explanation to understand the data-flow transformations. We refer to the sequentially constructive foundations [17] for a more formal and wider introduction.

The minimal SCL is adopted from C and Esterel. The concurrent and sequential control-flow of an SCL program is given by an SCG, which acts as an internal representation for elaboration, analysis and code generation. Rows two and three of Fig. 2 present an overview of SCL and SCG elements and the mapping between them. SCL is a concurrent imperative language with shared variable communication. Variables can be both *written* and *read* by concurrent threads. Reads and writes are collectively referred to as variable *accesses*. SCL programs consist of one or more sequentially ordered statements with the following *abstract syntax* of *statements*

$$s ::= x = e \mid s \,;\, s \mid \text{if } (e)\ s \text{ else } s \mid l\!: s \mid \text{goto } l \mid \text{fork } s \text{ par } s \text{ join} \mid \text{pause}$$

where x is a *variable*, e is an *expression* and $l \in L$ is a *program label*. The statements s comprise the standard operations assignment, the sequence operator, conditional statements, labelled commands and jumps.

The sublanguage of *expressions* e used in assignments and conditionals is not restricted. However, we rule out side effects when evaluating e. Our notion of sequential constructiveness is based on the idea that the compiler guarantees a strict "initialize-update-read" (*iur*) execution schedule during each macro tick. The *initialize* phase is given by the execution of a class of writes which we call *absolute* writes (e. g., "x = 1"), while the *update* phase consists of executing *relative* writes where scheduling order does not matter (e. g., "x += 2" and "x +=3" can be scheduled in any order, with the same result). All the *read* accesses, in particular the conditional statements which influence the control-flow, are done last.

2.1 SCG Representation

An SCG is a labelled graph $G = (N, E)$ whose *statement nodes* N correspond to the statements of the program, and whose edges E reflect the sequential execution ordering and data dependencies between the statements. Nodes and edges are further described by various attributes. A node n is labelled by the *statement type*. Nodes labelled with $x = e$ are referred to as *assignment nodes*, those with if (e) as *condition nodes*, all other nodes are referred by their statement type (*entry nodes*, *exit nodes*, etc.). Fig. 2 sketches how SCG elements correspond to an SCL program. A technical report [16] describes this mapping in detail.

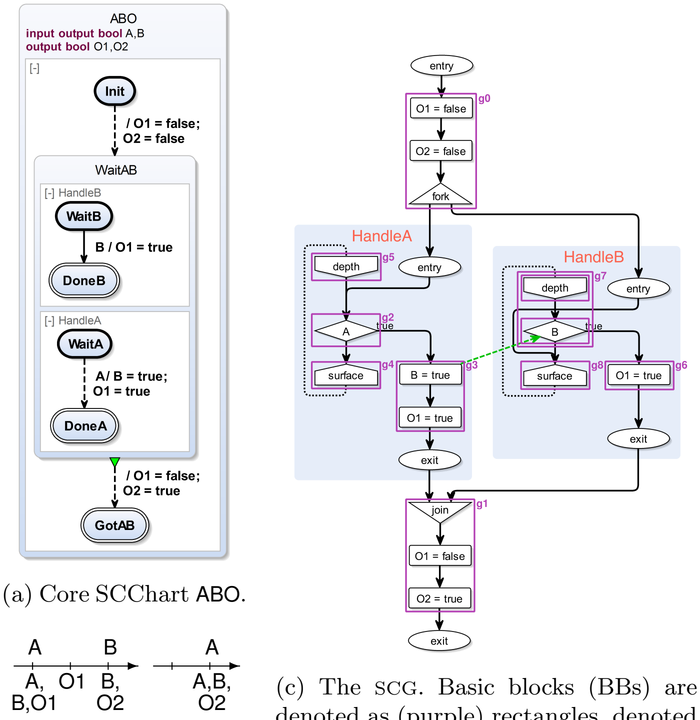
Every edge e has a type $e.type$ that specifies the nature of the particular ordering constraint expressed by e. Edges that follow the initialize-update-read schedule are labeled *iur*-edges. *iur*-edges combined with the sequential control-flow edges are termed *instantaneous* edges. An *SC-schedule* is a subset of instantaneous edges of an SCG. A *structural SC-schedule* is an *SC-schedule* that is solely derived by analysis of the program structure. A program for which the structural SC-schedule is acyclic is *structurally acyclic SC*, abbreviated *SASC*. The data-flow approach presented here requires that the SCG is SASC; this, for example, forbids any loops that are *instantaneous*, i. e., where the loop body is not interrupted by a tick boundary.

2.2 The ABO Example

The ABO example shown in Fig. 3a illustrates the concepts of core SCCharts, namely synchronous ticks, concurrency, deterministic scheduling of concurrent shared variable accesses, and sequential overwriting of variables.

The execution of an SCChart is divided into a sequence of logical ticks. The *interface declaration* of ABO states that A and B are boolean *inputs*, which are initialized by the environment at the beginning of each tick, as well as *outputs*, which are fed back to the environment at the end of each tick. O1 and O2 are boolean outputs, which here are initialized to false, and which are persistent from one tick to the next.

Initially, the system is in state WaitAB, which consists of regions (threads) HandleA and HandleB. HandleA stays in the initial state WaitA until the boolean input A becomes true. Then it sets B and O1 to true and transitions to state DoneA, which is final and hence terminates HandleA. Similarly, WaitB waits for



(a) Core SCChart ABO.

(b) Possible execution traces, with inputs above the tick time line and outputs below.

(c) The SCG. Basic blocks (BBs) are denoted as (purple) rectangles, denoted with their guards; g*i* guards BB *i*. The data dependence on B (dashed arrow) splits BB 7 into two scheduling blocks.

```
module ABO
input output bool A, B;
output bool O1, O2;
{
  O1 = false;
  O2 = false;

  fork
    HandleA:
    if (!A) {
      pause;
      goto HandleA;
    }
    B = true;
    O1 = true;

  par
    HandleB:
    pause;
    if (!B) {
      goto HandleB;
    }
    O1 = true;

  join;

  O1 = false;
  O2 = true;
}
```

(d) SCL code.

Fig. 3. The ABO example, illustrating the Core SCChart features [15].

B to become true, sets O1 to true, and transitions to final state DoneB. Once both HandleA and HandleB have terminated, WaitAB is left, O1 is set to false, O2 to true, and state GotAB is entered. The dashed edge denotes the transition to DoneA to be *immediate*, meaning that HandleA does not pause for a tick before it is ready to detect the transition trigger. In contrast, the transition to DoneB in HandleB is *delayed* and thus does not get triggered in any tick in which WaitB is entered.

Two possible execution traces are shown in Fig. 3b. The first trace begins with A set to true by the environment in the initial tick. This triggers the transition to DoneA and sets both B and O1 to true. As this is the initial tick, the non-immediate transition from WaitB to DoneB does not get triggered by the B. In the next tick, all inputs are false, no transitions are triggered, and O1 stays at true. In the third and last tick, B then triggers the transition to DoneB, which sets O1 to true, but sequentially afterwards, O1 is set to false again as part of the transition to GotAB, which is triggered by the termination of HandleA and

HandleB. Hence, at the end of this tick, only O2 will be true because the SCMoC allows O1 to be overwritten sequentially. The second trace illustrates how A in the second tick triggers the transitions to DoneA as well as to DoneB, hence emission of B and O2 and the termination of the automaton.

3 Data-Flow M2M Transformation

As depicted in Fig. 1 the transformation of the SCG mapped from a normalized SCChart to a sequentialized SCG is done in several distinct steps. Following the SLIC approach [10] every step is executed as an M2M transformation. This section explains each of these steps, namely dependency analysis, basic block arrangement, guard creation, scheduling, and sequentialization. For the subsequent analyses we assume that superfluous fork-join-constructs, i. e., containing only one thread, and *dead code* were removed.

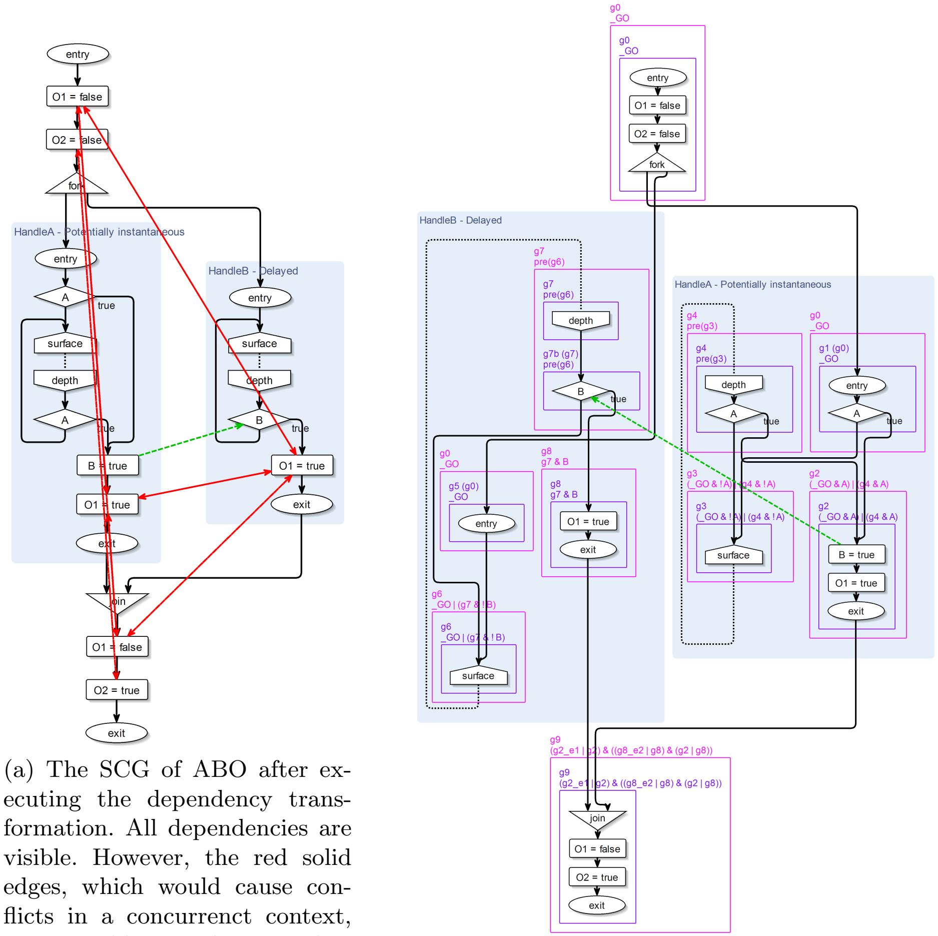
3.1 Dependency Analysis

The analysis of dependencies between different expressions is done straightforwardly. Every assignment and conditional in the program is checked for variable accesses. The type of the access is stored during the compilation. For each pair of accesses it is determined if the access is *concurrent* and/or *confluent*. Confluence means that the scheduling order does not matter, as is the case for example for the aforementioned relative writes.

All non-concurrent (confluent or non-confluent) dependencies are handled according to the sequential control-flow, whereas concurrent accesses are scheduled following the "initialize-update-read" protocol. According to Sec. 2.1, SCL programs that contain immediate dependency cycles are not *constructive* and are rejected.

In the ABO example applying the dependency analysis reveals one concurrent dependency as depicted in Fig. 4a. The concurrent dependency is shown as green, dashed line between the two threads connecting the B = true assignment in HandleA with the B conditional of HandleB. Furthermore, there are several other dependencies indicated by the red solid edges. These dependencies would cause conflicts in a purely concurrent context if they were not confluent. However, in ABO they can be executed sequentially. For example, the assignment to O1 at the top before the fork gets always executed before the assignment to O1 after the join.[1]

[1] Non-conflicting non-concurrent dependencies are normally not shown in our tool chain. Here, we activated the visualization to demonstrate the dependency analysis. However, concurrent non-confluent dependencies, which represent conflicts, are always shown.



(a) The SCG of ABO after executing the dependency transformation. All dependencies are visible. However, the red solid edges, which would cause conflicts in a concurrent context, are unproblematic because they either can be solved sequentially or are confluent.

(b) The SCG of ABO after proceeding with the BB and guard creation transformations. The nodes are encapsulated in their SB and, hence, BB. Each BB has its *guard expression* attached.

Fig. 4. Transformation of the SCG following the SLIC approach executing the dependency, the basic block, and the guard creation transformations.

3.2 Basic Block Arrangement

The data-flow compilation approach converts all control flow, be it sequential or concurrent, into a flat sequence of *guarded commands*. As a consequence, we cannot handle instantaneous loops with this code generation approach, as mentioned before. A guarded command is a statement that gets executed in the current tick if and only if a specific guard evaluates to true in the current tick; guards have a unique value throughout each tick.

To economize on the number of guards, we make use of the standard concept of Basic Blocks (BBs). In our setting BBs denote sets of statements that are executed together in a tick, i. e., either all or none of them are executed in a tick. Thus, all statements within a BB may share the same guard. The following rules, defined in a more formal way elsewhere [13], define BBs:

- A BB begins if the SCG representation of that statement has two or more incoming control-flow edges.
- A BB ends with a statement that forks the SCG control-flow and hence, has two or more outgoing control-flow edges. The last instruction of a thread may also be the closure of a BB.
- BBs are split at pause statements.
- SCG fork nodes close a BB, whereas join nodes start a new one.
- Any statement of a given program can only be included in one BB at any time.

The statements in a BB are not necessarily executed atomically, in the sense that BBs of concurrent threads may be interspersed with each other in order to satisfy dependencies induced by shared variables. Therefore, we may further divide each BB in Scheduling Blocks (SBs). With the BBs defining **what** set of instructions becomes active in the current tick, the SBs take the dependencies into account and define **when** a particular instruction set is executed, thus, defining the execution order. Therefore, a SB subdivides a BB if an incoming dependency edge targets an instruction inside the BB because the scheduler might want to reschedule here.

The arrangement of the blocks for the ABO example is depicted in Fig 4b. Building the blocks according to the rules imposed earlier in this section, each instruction is included in a SB directly surrounding it. The SB itself is included in a BB. In ABO most BBs only comprise one SB. A BB containing two SBs can be seen at the top of HandleB marked with the guard g7. This block gets divided due to the dependency found in Sec. 3.1.

3.3 Guard Creation

In every tick instance a BB may be *active* or *inactive*. The activity state of a BB depends on previous BBs and is called the *guard* of the BB. The BB determines the guard expression, whereas the SB defines the order of execution as described in Sec. 3.2.

Simple Guards directly depend on their predecessors. The guard of the first block in an SCL program depends on the *GO* start signal of the environment usually emitted at the *initialization* or *reset* of the program. Outgoing control-flows of a BB may serve as *guard expressions*, or *activators*, for succeeding BBs. Thus, a guard is a disjunction of all preceding activators and evaluates to true as long as one incoming activator is true. Generally speaking, a BB is active, if and only if at least one of its guard expressions is active in the same tick.

Not all BBs need an own guard if their order of activation is determined statically at compile-time. Therefore, we use standard compiler techniques such as *copy propagation* [2] to reduce the amount of needed unique guards.

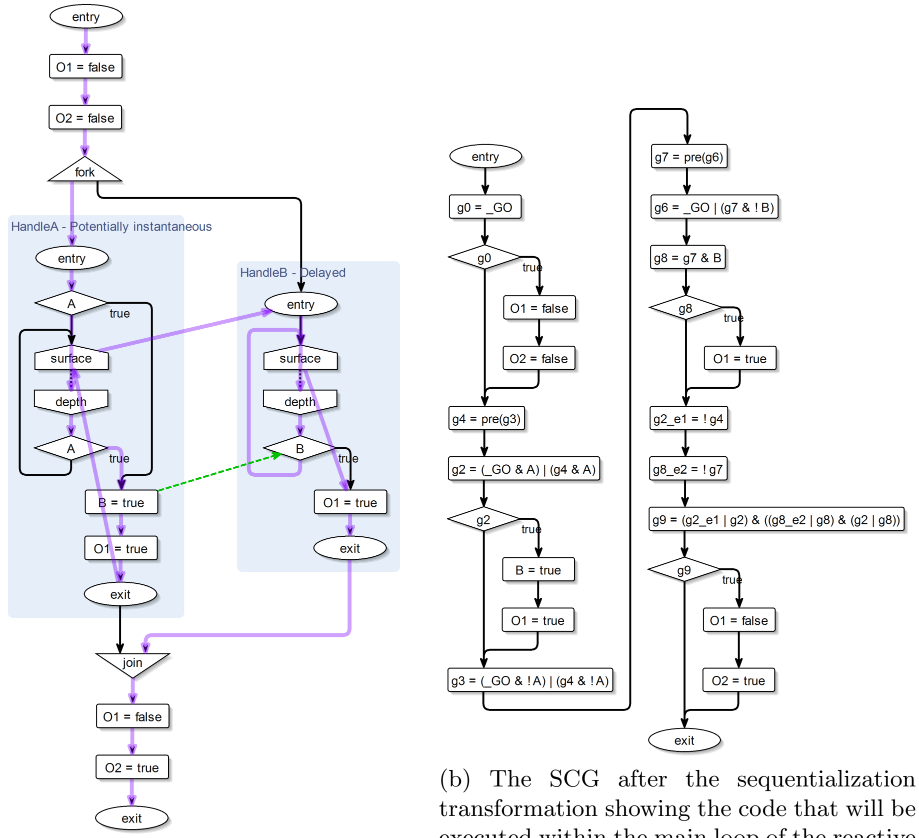
The guards generated for the ABO example are shown in Fig. 4b. Each BB has caption lines at the top. The first line shows the guard of the block and the guard expression generated for this block is displayed in the second line. For instance, the first block of ABO is named g0. Since it is the first block of the program, it gets activated once the program starts (and at every reset). Therefore, the guard expression is equal to the _GO signal of the environment. Another example is g8 in HandleB. Here, the guard depends on its predecessor g7. Furthermore, the BB of g8 is a successor of the *true branch* of the conditional in the block of g7 which evaluates to true if B is true. Therefore, the guard expression of g8 is g7 & B.

Notice that the BB of g7 includes two SBs due to the incoming dependency edge. SBs are named like their parent BB and suffixed alphabetically after the first one. As explained in Sec. 3.2, the scheduler may reschedule between any two SBs. However, as both SBs live inside the same BB, the guard expression of both SB are identical as explained earlier. The guard expression of g7 is pre(g6) meaning that it is set to the activation status of g6 in the tick before. This implements a *tick boundary*, and we also say that a program resumes the execution at the *depth* here if it was paused at the *surface* in the preceding tick.

Also, the effects of the copy propagation can be seen in the figure. SB g1 in HandleA will immediately become active after program activation because it solely depends on g0. Hence, the _GO signal also triggers this block.

Complex Guards cannot simply depend on their direct predecessors. In SCL programs, forked threads must be joined at some point in time. The join instruction will not proceed unless each thread has finished. Hence, a BB including a join node only activates if all preceding threads are terminated and at least one of them exited in the actual tick instance. Each thread status is signaled by an *empty flag* which describes whether or not a thread is inactive. All empty flags are combined in a conjunction together with a combination of exit codes that signal whether at least one thread terminated in this tick instance. The empty flag is combined with the _GO signal of the preceding circuit to detect active instantaneous threads. BBs that are responsible for joining threads are also called *synchronizer*. The construction of the synchronizer is done similar to the synchronizer circuit described in Compiling Esterel [11].

In the ABO example the BB with guard g9 is activated by a complex guard expression. The conjunction consists of three parts. The first two parts (g2_e1 | g2) & (g8_e2 | g8) indicate if the threads HandleA and HandleB are inactive or terminated in this tick. g2 and g8 are the SBs that are active if their thread is exited in this tick because they include the corresponding exit node. The empty flags, suffixed with _*ex*, are set to true if the registers, i. e., pause instructions, are inactive. To activate the block with the join node, at least one thread must have been exited in this tick. Therefore the third part (g2 | g8) is checked.



(a) The SCG after the scheduling transformation depicting the route the scheduler has chosen. Here, only one context switch is necessary.

(b) The SCG after the sequentialization transformation showing the code that will be executed within the main loop of the reactive tick cycle.

Fig. 5. Transformation of the SCG following the SLIC approach executing the scheduling and sequentialize transformations.

3.4 Scheduling

In the scheduling step (cf. Fig. 1) the transformation returns a valid schedule for the given program if one exists. Therefore, the blocks, and hence their included instructions, are ordered topologically according to their guard expressions and with respect to any dependencies between the blocks. This may result in arbitrarily many context switches between threads. However, to support any low-level analysis on threads that may be executed later on, it is desirable to perform as few switches between threads as possible. One example for a low-level analysis is a worst-case execution time (WCET) analysis. Hints about superfluous context switches can be found in our work towards interactive timing analysis [5]. Here, any context switch results in the insertion of an so called *timing program points*. As these points may influence measurements, one should not include more than necessary.

The schedule chosen for ABO can be seen in Fig. 5a. The bold purple arrow depicts the path chosen as schedule. As one would expect the program starts at the first entry node. Then, at the fork, the scheduler chooses to proceed with HandleA. It switches to HandleB not until HandleA has finished and proceeds to the join after both threads completed.

This is not the only possible schedule. Nevertheless, every valid schedule will produce the same deterministic output [17]. It would have been valid to start with HandleB and then switch to HandleA. However, the scheduler could not have finished HandleB completely as it has done with HandleA because HandleB depends on HandleA due to the conditional node testing B. Therefore, beginning the other way around would lead to more context switches.

3.5 Sequentializing

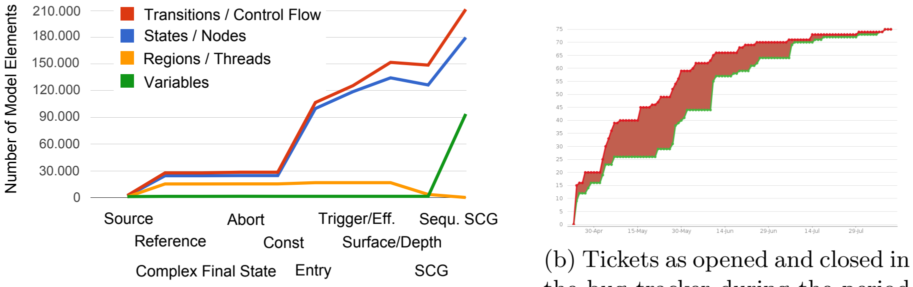
Finally, from the schedule we can derive the sequentialized program. Therefore, the guards created before are written in the order defined by the scheduler. If an SB contains assignments they must be executed if the BB is active. Hence, these are added to the sequentialized program guarded by the guard of their block.

The fully sequentialized program for the ABO program is shown in Fig. 5b. One will recognize the guard expressions from Fig. 4b. Here, they are ordered according to the schedule depicted in Fig. 5a. Whenever a guard is responsible for any assignments, a conditional is added which holds the guard as condition. For instance, at the beginning of the program g0 is set to the _GO signal of the environment. Hence, it will be true in the first tick and therefore O1 and O2 will be initialized with false. If A is also true in the first tick, B and O1 are set to true. However, as described in Sec. 2.2 the join g9 is not activated because HandleB cannot reach its exit node in g8 in the first tick. g8's expression is g7 & B and g7 depends on pre(g6). Thus, g6 must have been active in the tick before so that HandleB may terminate in the actual tick. This is not possible in the first tick.

Fig. 5b also depicts the empty flags g2_e1 and g8_e2 needed for the synchronizer. As discussed in Sec. 3.3 these indicate whether a thread is inactive in the actual tick. Therefore, they negate the status of the concurrent depth nodes which abstractly resemble registers. The complex guard expression is then constructed as explained earlier in Sec. 3.3. The sequentialized program is now ready to be deployed. It can be translated to various languages such as C, Java or VHDL. Deployment to hardware requires the final deployment step to apply an SSA transformation to the variables used in the program if they are written more than once [8].

4 Experimental Results

In summer term 2014 we launched a student project [14] where the task was to build an SCCharts based controller for a rather large model railway system.



(a) Hiding complexity by using Extended SCCharts features: Expansion of SCCharts features down to sequentialized SCG elements gives an idea of the complexity of this model [14].

(b) Tickets as opened and closed in the bug tracker during the period of this project validating maintainability and extendability of the model based SLIC compiler approach [14].

Fig. 6. The SCCharts SLIC based compilation approach turns out to be practically usable even for complex models and to be maintainable and extendable.

The model railway system consists of a total track length of 127 meters split into 48 individually controllable block segments with 28 switch points. The final controller is able to drive 11 trains simultaneously with integrated dead-lock and live-lock avoidance. The controller fully expands to 135,000 states, 152,000 transitions and 17,000 concurrent regions after eliminating all reference states by a reference state compiler transformation. 1,628 states were modeled manually together with 2,219 transition and 183 concurrent regions. Compared to David Harel's Wristwatch [7], which was considered a complex statechart back in 1986, we would also call the SCCharts model railway controller at least a medium-size real-world complex system. It compiles using the presented tool chain in 2-3 minutes and generates about 650,000 lines of C code. Still the response time of the running controller was measured to be smaller than 2ms on a standard PC.

We measured the number of model elements for the SCCharts model railway controller example at every intermediate stage of the SLIC compile chain (cf. Fig. 1). Fig. 6a shows the result and suggests how much complexity of the resulting sequentialized SCG model could be hidden by using Extended SCCharts features for modeling the complex behavior of this controller. The students were not only using our SCCharts compiler tool chain but also struggling with teething troubles of our early prototype compiler. That resulted in many bug reports especially in the middle of the project when the students started modeling. As Fig. 6b attests we were able to quickly resolve most of the problems without introducing more new problems. This circumstance validates maintainability of the model based SLIC approach used for the compiler. Additionally new feature requests like reference state expansion arose during the project and could be integrated into the existing compiler validating extendability of our overall approach.

5 Conclusions

As discussed before [10] being able to quickly prototype a modular compiler that is easy to validate and to customize prompted us to follow the SLIC approach. The SLIC approach showed that it is possible to use model-driven aspects to build a compiler that is also fairly compact and efficient. Following this route we here presented SLIC transformation rules for the data-flow approach, one of the two proposed low-level methods for generating code for SCCharts [15]. Similar to the high-level SCCharts transformation rules, the SCL transformations obey the proposed pattern:

- The compilation steps are M2M transformations where the resulting model contains all information. There are no other, hidden data structures.
- The intermediate transformation steps are in the same language. We just apply a sequence of language operations, that added one analyses result at a time.

We see numerous directions for future work. For example, we want to use existing simulation tools to evaluate the activity state of each guard during runtime. Hence, feeding the information back to the simulator and using the M2M transformation information of the SLIC approach, it should be possible to display each active area of the SCCharts on modelling level. Synchronizing threads as explained in Sec. 3.3 becomes difficult when dealing with instantaneous feedback loops. There are possibilities to handle such a feedback in the data-flow approach as long as at least one thread is delayed to prevent instantaneous cycles. We would like to further investigate possibilities to handle feedbacks and implement these in our tool chain. Another active area is that of interactive timing analysis [6], where we investigate how to best preserve timing-information across M2M transformations. The main advantage of our approach is its interactivity. Nonetheless we envision a fully automatic compilation process including the possibility to include our compiler in scripts (e. g., a Makefile) or using it online in the Web. Another important question is how much parallelism should we derive from the initial concurrency modeled by the modeler. Allowing programs to be executed partly in parallel without full sequentialization is ongoing research.

References

1. J. Aguado, M. Mendler, R. von Hanxleden, and I. Fuhrmann. Denotational fixed-point semantics for constructive scheduling of synchronous concurrency. *Acta Informatica, Special Issue on Combining Compositionality and Concurrency*, 52(4):393–442, 2015.
2. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers - Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachusetts, 1986.
3. A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The Synchronous Languages Twelve Years Later. In *Proc. IEEE, Special Issue on Embedded Systems*, volume 91, pages 64–83, Piscataway, NJ, USA, Jan. 2003. IEEE.

4. S. A. Edwards. Tutorial: Compiling concurrent languages for sequential processors. *ACM Transactions on Design Automation of Electronic Systems*, 8(2):141–187, Apr. 2003.
5. I. Fuhrmann, D. Broman, S. Smyth, and R. von Hanxleden. Towards interactive timing analysis for designing reactive systems. Reconciling Performance and Predictability (RePP'14), satellite event of ETAPS'14, Apr. 2014.
6. I. Fuhrmann, D. Broman, S. Smyth, and R. von Hanxleden. Towards interactive timing analysis for designing reactive systems. Technical Report UCB/EECS-2014-26, EECS Department, University of California, Berkeley, Apr. 2014.
7. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
8. G. Johannsen. Hardwaresynthese aus SCCharts. Master thesis, Kiel University, Department of Computer Science, Oct. 2013. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/gjo-mt.pdf.
9. E. A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, 2006.
10. C. Motika, S. Smyth, and R. von Hanxleden. Compiling SCCharts—A case-study on interactive model-based compilation. In *Proceedings of the 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2014)*, volume 8802 of *LNCS*, pages 443–462, Corfu, Greece, Oct. 2014.
11. D. Potop-Butucaru, S. A. Edwards, and G. Berry. *Compiling Esterel*. Springer, May 2007.
12. K. Rathlev, S. Smyth, C. Motika, R. von Hanxleden, and M. Mendler. SCEst: Sequentially Constructive Esterel. In *Proceedings of the 13th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE'15)*, Austin, TX, USA, Sept. 2015.
13. S. Smyth. Code generation for sequential constructiveness. Diploma thesis, Kiel University, Department of Computer Science, July 2013. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/ssm-dt.pdf.
14. S. Smyth, C. Motika, A. Schulz-Rosengarten, N. B. Wechselberg, C. Sprung, and R. von Hanxleden. SCCharts: the railway project report. Technical Report 1510, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Aug. 2015. ISSN 2192-6247.
15. R. von Hanxleden, B. Duderstadt, C. Motika, S. Smyth, M. Mendler, J. Aguado, S. Mercer, and O. O'Brien. SCCharts: Sequentially Constructive Statecharts for safety-critical applications. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*, Edinburgh, UK, June 2014. ACM.
16. R. von Hanxleden, M. Mendler, J. Aguado, B. Duderstadt, I. Fuhrmann, C. Motika, S. Mercer, O. O'Brien, and P. Roop. Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation. Technical Report 1308, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Aug. 2013. ISSN 2192-6247.
17. R. von Hanxleden, M. Mendler, J. Aguado, B. Duderstadt, I. Fuhrmann, C. Motika, S. Mercer, O. O'Brien, and P. Roop. Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation. *ACM Transactions on Embedded Computing Systems, Special Issue on Applications of Concurrency to System Design*, 13(4s):144:1–144:26, July 2014.

Bytecode-Generierung eines neuartigen Java Compilers

Evelyn Fikus, Franziska Fütterling, Julia Schubert, Andreas Stadelmeier,
Martin Plümicke

Duale Hochschule Baden-Württemberg,
Jägerstraße 56, 70174 Stuttgart, Deutschland
{evelyn.fikus,franziska.fuetterling,julia.schubert}@hp.com
a.stadelmeier@hb.dhbw-stuttgart.de
pl@dhbw.de
http://www.dhbw-stuttgart.de/home/

Zusammenfassung. In Java war es bisher nötig alle Typen von Methoden explizit anzugeben. Oftmals lassen sich Typen aber auch ohne direkte Angabe berechnen. Dazu haben wir ein System entwickelt, das bisher nur die Typen berechnen konnte und Java-ähnlichen Code erzeugt hat. Um den Code zu kompilieren, musste der Code so verändert werden, dass insbesondere die Klassenparameter verändert werden, was dazu führte, dass alle Deklarationen und new-Aufrufe ebenfalls angepasst werden mussten. Durch die nun eigenständige Bytecode-Generierung ist eine durchgängige Compilation in unserer Umgebung möglich.

1 Einleitung

Im Rahmen mehrerer Forschungsarbeiten wurde ein neuartiger Java Compiler entwickelt, der Typinferenz in Java ermöglicht[1]. Mithilfe von Algorithmen wird dabei der Typ eines gegebenen Ausdrucks rekonstruiert, sodass auf die explizite Angabe von Typen verzichtet werden kann[2].

Im Jahr 2013 wurde der von dem neuartigen Java Compiler eingesetzte Typinferenzalgorithmus erweitert und auf die neueste Java Version 8 angepasst. Ein zentrales Feature, das von Java 8 neu eingeführt wurde, stellen Lambda-Ausdrücke dar. Hierbei handelt es sich um ein Sprachkonstrukt, das von der funktionalen Programmierung inspiriert ist[3]. Dadurch wird in Java die Möglichkeit geschaffen, namenlose Methoden zu definieren und diese einer Variable zuzuweisen, die anschließend beispielsweise auch als Parameter an eine andere Methode übergeben werden kann[4].

Unter Verwendung des entwickelten, neuartigen Compilers wird angestrebt, die Schreibweise und die Syntax der Lambda Ausdrücke noch stärker an das funktionale Programmierparadigma, wie es zum Beispiel in Haskell vorzufinden ist [5], anzunähern. Dazu soll innerhalb der Lambda-Ausdrücke ebenfalls Typinferenz zum Einsatz kommen.

Nachfolgend wird ein Code-Beispiel dargestellt, das zunächst aufzeigt wie dessen Implementierung in Java 8 aussieht und anschließend demonstriert welche Vereinfachungen durch den neuartigen Compiler erreicht werden sollen:

Beispiel in Java 8:

```
class Bytecode<B> {
  Function<B, B> id = (B x) -> x;
}
```

Beispiel mit Vereinfachungen durch neuartigen Compiler:

```
class Bytecode {
  id = x -> x;
}
```

Diese Schwachpunkte werden durch den neuartigen Compiler und der darin mitgelieferten Typinferenz beseitigt. Wie in der zweiten Umsetzung zu sehen ist, entfällt bei dessen Einsatz die explizite Angabe der generischen Typen. Bei der Verwendung des Lambda-Ausdrucks wird durch den Typinferenzalgorithmus automatisch der korrekte Typ inferiert und eingesetzt. Dabei wird zusätzlich ermöglicht, dass für jeden Ausdruck individuell ein variabler Typ vergeben werden kann. Der von der Typinferenz erzeugte Code sieht demnach folgendermaßen aus:

```
class Bytecode {
  <B> Function<B,B> id = (B x) -> x;
}
```

Die zugrundeliegende Motivation ist es, das in Haskell charakteristische Konstrukt der Typvariablen zur Definition von Funktionen, die für alle möglichen Variablentypen gültig sind, in Java abzubilden. Es soll somit realisiert werden, dass stets mit einem möglichst allgemeinen Typen gearbeitet wird. Dadurch wird schließlich eine dynamische Handhabung von variablen Typen in Java ermöglicht.

Ein erster Ansatz wäre den nach Typinferenz erzeugten Java Code durch den Standard Java-Compiler zu übersetzen. Da der Standard-Java aber keine generischen Typvariabeln für Attribute kennt, ist auf alle Fälle eine Code-Transformation nötig. Man könnte die Typvariablen als generische Typen der Klasse definieren, was den Nachteil hätte, dass alle new-Aufrufe angepasst werden müssten. Oder man könnte die generischen Typvaraibeln durch Object ersetzen. Wir haben uns für die direkte Erzeugung von Bytecode entschieden, auch weil zukünftige Erweiterungen, dann einfacher zu realisieren sind.

2 Ähnliche Arbeiten

Es gibt verschiedene Bibliotheken, die der Analyse, Manipulation und Erzeugung von Bytecode dienen. Für die Generierung des Bytecodes innerhalb des neuartigen Compilers wurde die Nutzung unterschiedlicher Bibliotheken abgewogen.

Eine der in Erwägung gezogenen Bibliotheken ist Jasmin[6]. Allerdings hat sich herausgestellt, dass diese Bibliothek nicht für die Erzeugung des Bytecodes geeignet ist. Die Ursache liegt darin, dass der zu übersetzende Code in der

Jasmin spezifischen Syntax vorliegen müsste, aus der schließlich der Bytecode erzeugt wird. Da der Code innerhalb des neuartigen Compilers nicht in der Jasmin Syntax vorliegt, konnte diese Bibliothek nicht verwendet werden.

Außerdem wurde ebenfalls das Framework ASM in Betracht gezogen, um die Erzeugung des Bytecodes innerhalb des neuartigen Compilers zu realisieren[7]. ASM zeichnet sich dadurch aus, dass es besonders schnell beim Parsen des Bytecodes ist[8]. Die Nutzung dieses Frameworks ist darauf ausgerichtet, dass auf einer tiefen Ebene ohne Abstrahierungsgrad gearbeitet wird. Dafür stellt ASM viele Funktionen bereit.

Eine weitere Bibliothek, die für die angestrebte Generierung von Bytecode in Frage kommt, ist BCEL[9]. Charakteristisch für BCEL ist, dass vorgefertigte Konstrukte mitgeliefert werden, die von der konkreten Erzeugung des Bytecodes abstrahieren. Ein Beispiel dafür stellt die InstructionFactory dar[10]. Dadurch muss der Entwickler nicht auf der untersten Ebene arbeiten und der Einstieg in die Nutzung der Bibliothek wird somit vereinfacht.

Schlussendlich wurde die Bibliothek BCEL für die Erzeugung des Bytecodes im neuartigen Java Compiler ausgewählt, da die Geschwindigkeitsvorteile, die ASM bieten würde, in diesem Fall nicht die höchste Priorität haben. Ausschlaggebend waren vielmehr die von BCEL zur Verfügung gestellten abstrakten Konstrukte, die die Bytecode-Generierung insgesamt einfacher gestalten. Dadurch sollte eine schnelle und erfolgreiche Einarbeitung begünstigt werden.

3 Bytecode-Generierung mit BCEL

In Java wird Quellcode mit Hilfe des Compilers in binäre Klassendateien umgewandelt. Diese Klassendateien werden auch als Bytecode bezeichnet. Oftmals soll in bereits bestehende Javaprogramme eingegriffen werden, um beispielsweise eine Performanzsteigerung herbeizuführen oder Java parametrisierte Typen hinzuzufügen. Um diese Aspekte zu realisieren, muss der bestehende Java Compiler oder die Java Virtual Machine verändert werden. Ein viel effizienterer und plattformunabhängiger Weg ist, den vom Compiler erzeugten Bytecode manuell anzupassen. Dies ist ein Weg, den viele Informatiker gehen. Allerdings sind deren Lösungen oft projektabhängig und selten wiederverwendbar.

Die Byte Code Engineering Library (BCEL)[9] ist ein von der Apache Organisation gesponsertes Projekt. BCEL wird von Apache selbst als Werkzeugsatz bezeichnet, welcher Klassen und Schnittstellen zur statischen Analyse und Veränderung von Java Bytecode zur Verfügung stellt.

BCEL stellt drei Pakete zur Verfügung: Eines, welches Klassen beinhaltet, die statische Einschränkungen der Klassendateien realisieren und nicht für Bytecode Modifikationen gedacht sind. Eines, welches erlaubt, dynamisch JavaClass- oder Methodenobjekte zu generieren und modifizieren. Und das letzte, welches diverse Code Beispiele und Werkzeuge, beispielsweise zum Anzeigen von Klassendateien, bereitstellt. Die für dieses Projekt vorrangig verwendeten Teile der Bibliothek sind ClassGen, MethodGen und diverse InstructionHandles. Sie bilden die Abstraktionsebene zwischen Abstrakter Syntax und dem endgültigen

Bytecode. Sie speichern alle zur Bytecodegenerierung benötigten Informationen. Die Eintragung dieser Informationen in eine Classfile übernimmt im Anschluss das Framework. So kann BCEL beispielsweise den Konstantenpool eigentständig aus den gegebenen Informationen generieren. Im Folgenden werden die verwendeten Klassen genauer betrachtet:

- ClassGen liefert eine abstrakte Sicht für die dynamische Kreation und Transformation von Klassendateien. Hauptsächlich ist ClassGen für die Einhaltung der Beschränkungen von Java, wie die fest gecodeten *generic* Bytecode Adressen, zuständig. Auch MethodGen und ConstantPoolGen für die Realisierung von jeweils Methoden und Konstantenpool, sind in diesem Teil der Bibliothek zu finden. Folgende Abbildung 1 zeigt ein UML Diagramm des ClassGen der Bibliothek.

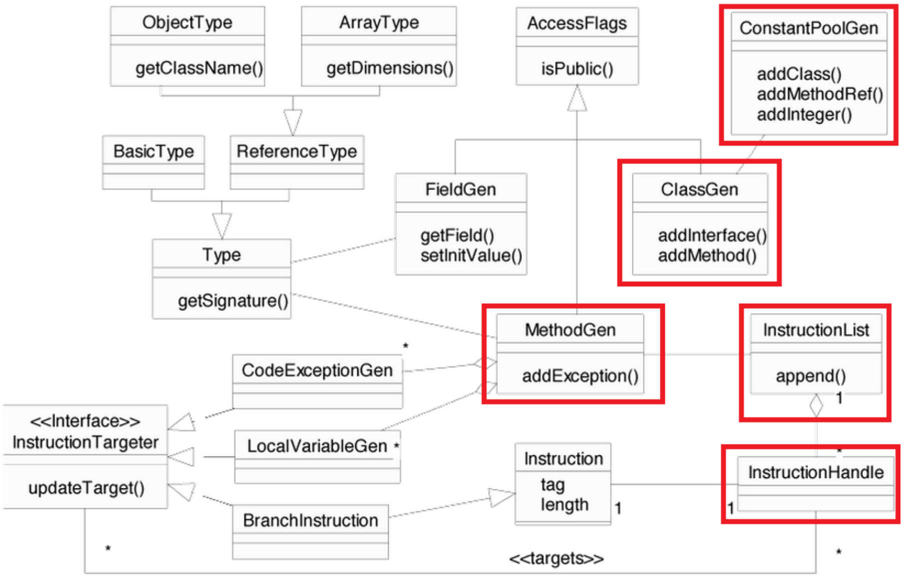


Abb. 1. UML of ClassGen[9]

- MethodGen wird verwendet, um Methoden der zu kompilierenden Klasse dem ClassGen hinzuzufügen. Des Weiteren vereint MethodGen alle vorhandenen InstructionHandles zu einer InstructionList.
- Viele der InstructionHandles stammen ursprünglich aus der InstructionFactory. Dieser Teil der Bibliothek soll die Generierung von Variablen, unter anderem von Konstanten, vereinfachen, indem bestimmte Instruktionen bereits vorgegeben werden. Die InstructionHandles an sich bilden den in diesem Projekt kleinsten Teil der BCEL Bibliothek. Sie werden für die Generierung von Bytecode für lokale sowie globale Variablen verwendet und schreiben

entsprechende Variablen mit Befehlen wie beispielsweise *iconst_1 (schreibt die Konstante 1)* in den Konstantenpool.

4 Vorgehensweise mit Hilfe von BCEL

Im Folgenden soll nun die Implementierung einer Methode zur Bytecode-Generierung innerhalb des neuartigen Java Compilers betrachtet und erläutert werden.

Im Syntaxbaum des neuartigen Compilers ist eine ähnliche Struktur zu finden, welches eine analoge Übertragung und Implementierung von BCEL ermöglicht. Zunächst wird die übergreifende Klasse Class.java des Compilers bearbeitet. Auf der gleichen Ebene in BCEL findet sich das ClassGen, welches innerhalb dieser Klasse den Hauptbestandteil für die Bytecode-Generierung bildet. Es wird zunächst also eine *genByteCode*- Methode implementiert, welche in der Lage ist, das ClassGen in eine Klassendatei zu schreiben. Dafür wird ein neues ClassGen deklariert, welches die dafür typischen Parameter, wie im vorigen Kapitel beschrieben, enthält. Diese Parameter sind dynamisch gestaltet, sodass bei Nutzung des Compilers die korrekten Daten des Programms des Anwenders in die Paramter eingetragen werden. Des Weiteren wird ein Konstantenpool erzeugt, welcher mit den Daten aus ClassGen gefüllt wird. Das ClassGen selbst wird an die unteren Hierarchieebenen des Compilers weitergegeben. Innerhalb dieses Compilers besteht die nächste Hierarchieebene aus Feldern. Entsprechend werden diese iteriert, sodass das ClassGen in allen Feldern gefüllt wird und wieder an die Class.java übergeben wird, welche als Ergebnis eine komplette Klassendatei zurückgibt. Nachfolgende Abbildung 2 soll den Zusammenhang zwischen den im Folgenden erläuterten Klassen veranschaulichen.

Die Felder, die innerhalb dieses Projektes die interessantesten sind, sind Methoden, umgesetzt in Method.java, beziehungsweise Konstruktoren. Der Parser des neuartigen Compilers unterscheidet nicht zwischen Methode und Konstruktor. Aus diesem Grund wird eine Methode, sollte sie die Kriterien eines Konstruktors erfüllen, nach dem Parsen in Constructor.java zu einem Konstruktor umgewandelt. Constructor.java erbt entsprechend von Method.java. Also muss in Method.java nun eine *genByteCode*-Methode implementiert werden. Die *genByteCode*-Methode in Method.java sieht wie folgt aus: Als Übergabeparameter wird die ClassGen übergeben. Dieses ClassGen ist das in Class.java erstellte, welches durch Method.java gefüllt wird. Der Konstantenpool des ClassGen wird lokal gespeichert. Es werden eine InstructionFactory, welche das ClassGen und den Konstantenpool übergeben bekommt, und eine InstructionList erzeugt.

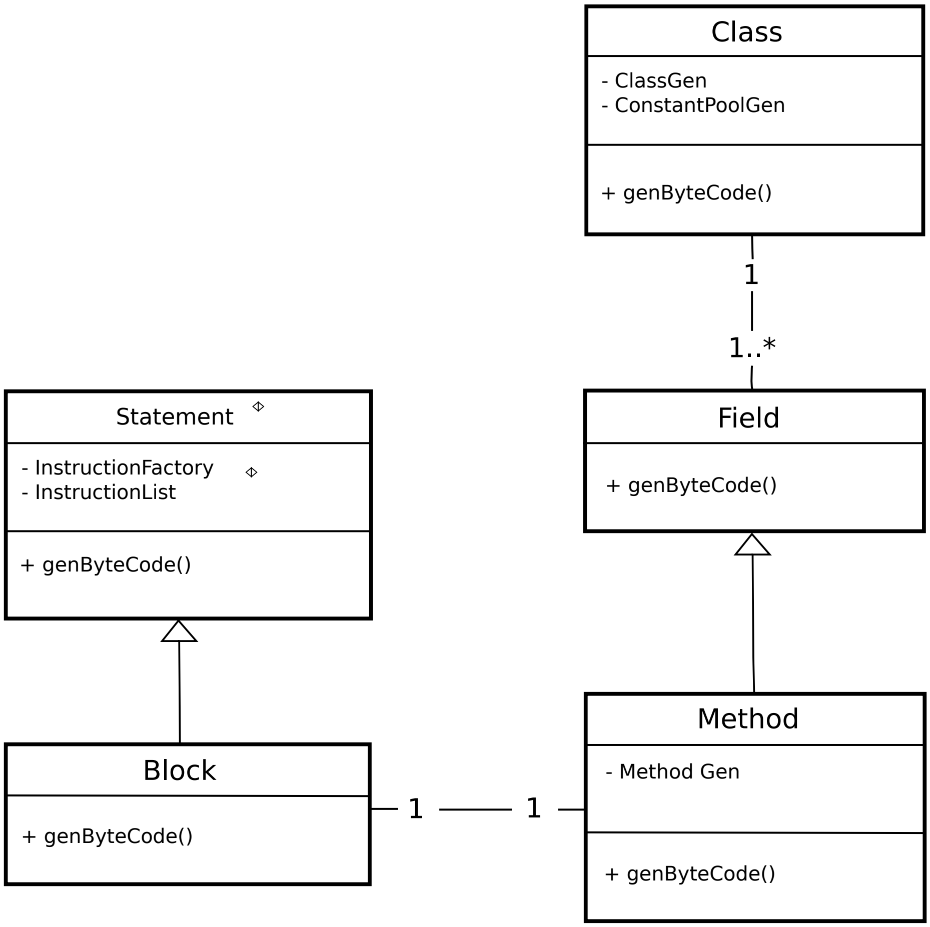


Abb. 2. Klassendiagramm

Diese werden im Verlauf der Methode verwendet beziehungsweise gefüllt. Anhand eines Objektes der Klasse Class.java kann im weiteren Verlauf der Name der Klasse, zu welcher die zu kompilierende Methode gehört, ausgelesen werden. Dies geschieht innerhalb der Erzeugung eines MethodGen. MethodGen braucht als Parameter die Access Flags, den Rückgabewert der Methode, deren Argumente, den Namen der Methode, den Namen der Klasse, zu welcher die Methode gehört, die InstructionList und den Konstantenpool. Eine Methode besitzt immer mindestens einen Block, welcher durch geschweifte Klammern gekennzeichnet ist. Der zur Methode zugehörige Block wird im nächsten Schritt in eine lokale Variable des Typs Block gespeichert. Anhand dieses Blocks kann die InstructionsList gefüllt werden, indem die *genByteCode*-Methode auf dem Block aufgerufen wird und das ClassGen dabei übergeben. Die resultierende InstructionList wird nun der Methode angefügt mit Hilfe des *append*-Befehls. Außerdem wird geprüft, ob der Block überhaupt irgendwelche Statements enthält. Sollte dies nicht der Fall sein, oder aber das letzte verfügbare Statement keine Instanz des Return-Typs besitzen, wird hier manuell ein void (leer) zurückgegeben und an der InstructionList angefügt. Anschließend wird noch ein dynamischer Stack für diese Methode gesetzt, welcher seine Größe durch die InstructionList bestimmt. Zuletzt wird die Methode dem ClassGen hinzugefügt. Folgender Code Abschnitt zeigt die eben erläuterte Implementierung zunächst in Class.java und anschließend in Method.java.

```
/* genByteCode in Class.java
*/
private InstructionFactory _factory;
private ConstantPoolGen    _cp;
private ClassGen           _cg;

    public ByteCodeResult genByteCode() throws IOException {

      _cg = new ClassGen(name, superClass.get_Name(), name + ".java",
          Constants.ACC_PUBLIC , new String[] {  });
      _cp = _cg.getConstantPool();
      _factory = new InstructionFactory(_cg, _cp);

      for(Field field : this.fielddecl){
          field.genByteCode(_cg);
        }

      ByteCodeResult code = new ByteCodeResult(_cg);
    return code;
    }
/* genByteCode in Method.java
*/
@Override
  public void genByteCode(ClassGen cg) {
    ConstantPoolGen _cp = cg.getConstantPool();
    InstructionFactory _factory = new InstructionFactory(cg, _cp);
    InstructionList il = new InstructionList();
    Class parentClass = this.getParentClass();

    MethodGen method = new MethodGen(Constants.ACC_PUBLIC, this.getType().
         getBytecodeType(), org.apache.bcel.generic.Type.NO_ARGS , new String
         [] {  }, this.get_Method_Name(), parentClass.name, il, _cp);

    Block block = this.get_Block();
    InstructionList blockInstructions = block.genByteCode(cg);

    il.append(blockInstructions);//Die vom Block generierten Instructions an die
         InstructionList der Methode anfügen

    if (block.get_Statement().size() == 0) { il.append(_factory.createReturn(
         org.apache.bcel.generic.Type.VOID)); }
    else {
        if (!(block.get_Statement().lastElement() instanceof Return)) { il.
             append(_factory.createReturn(org.apache.bcel.generic.Type.VOID));
              }
    }

    method.setMaxStack(); //Die Stack Größe automatisch berechnen lassen (erst nach dem
         alle Instructions angehängt wurden)

    cg.addMethod(method.getMethod());

  }
```

Nun wurden bereits zwei höhere Hierarchieebenen des Compilers betrachtet. Der bereits erwähnte Block bildet die nächst niedrigere Hierarchieebene. Blöcke werden innerhalb des Compilers in Block.java umgesetzt. Blöcke sind selbstständig nicht existenzfähig, da leere geschweifte Klammern in Java keine Bedeutung besitzen. Entsprechend müssen diese gefüllt werden - Mit Hilfe von Ausdrücken, welche im Compiler durch Statements ausgedrückt sind. Block erbt von Statement und besitzt eine eigene *genByteCode*-Methode, welche eine InstructionList zurückgibt, die durch die Iteration über Statement gefüllt wird. Statement bildet den in diesem Projekt kleinsten, bearbeiteten Bestandteil des Compilers und damit auch die niedrigste, bearbeitete Hierarchieebene.

Damit die Abläufe der Interaktionen zwischen den in Abbildung 2 dargestellten Klassen klar werden, stellt nachfolgende Abbildung 3 ein Sequenzdiagramm dieser Klassen zur Verfügung.

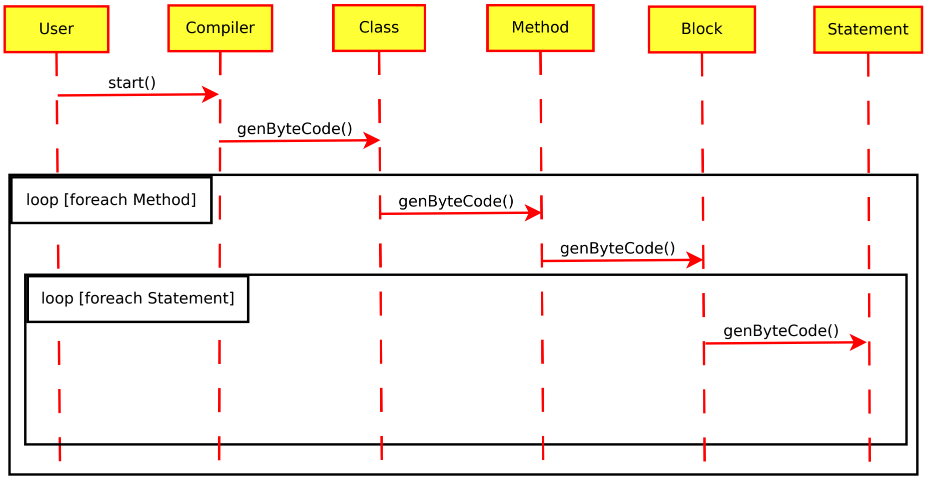


Abb. 3. Sequenzdiagramm

5 Fazit und Ausblick

Zum Zeitpunkt dieser Arbeit lässt sich abschließend sagen, dass ein Grundgerüst zur Vervollständigung und Erweiterung der Bytecode-Generierung erstellt wurde. Dieses Grundgerüst beinhaltet die Existenz einer Methode zur Bytecode-Generierung in allen dazu notwendigen Klassen. Aufgrund der zeitlichen Begrenzung des vorliegenden Projektes sowie der Unvollständigkeit der Parser-und Typinferenz-Funktionalitäten, auf die die Bytecode-Generierung aufbaut, sind nicht alle dieser Methoden funktionsfähig.

Zukünftige Projekte können auf dieser Arbeit aufbauen. Anpassungen am ausgegebenen Bytecode des Compilers sind nun leicht möglich. Zuvor konnte der neuartige Compiler zwar Typen im Java Quellcode inferieren und einsetzen, aber nicht eigentständig Bytecode erzeugen. Diesen Schritt musste ein Standard Java Compiler übernehmen wodurch kein Einfluss auf das Kompilat möglich war.

Dies ermöglicht unter anderem die Einführung von echten Generischen Typen in den Bytecode [13].

Quellenverzeichnis

1. Stadelmeier, A. and Plümicke, M., *Implementierung eines Typinferenzalgorithmus für Java 8*, in *Tagungsband des 17. Kolloquiums Programmiersprachen und Grundlagen der Programmierung (KPS'13)*, Ausgabe 2014/02, Seiten 127–134, *http://www.ba-horb.de/∼pl/papers/Typinferenz-Java8.pdf*, University Halle-Wittenberg, Institute of Computer Science 2014

2. Ehlers, C., *Typinferenz*, *http://www.fh-wedel.de/∼si/seminare/ws04/Ausarbeitung/4.Typcheck/staTyp4.htm*, o.J.

3. Inden, M., *Buch Java 8 - Die Neuerungen*, *Vgl. S. 3*, dpunkt.verlag GmbH 2014

4. Stadelmeier, A., *Java type inference as an Eclipse plugin*, in *Proceedings of the Studierendenkonferenz Informatik 2015*, 2015

5. Inden, M., *Buch Java 8 - Die Neuerungen*, *Vgl. S. 174*, dpunkt.verlag GmbH 2014

6. Meyer, J., *Jasmin Home Page*, *http://jasmin.sourceforge.net/*, Jasmin Home Page, 2004

7. OW2 Consortium, *ASM*, *http://asm.ow2.org/*, ASM, 2015

8. Bruneton, E. and Lenglet, R. and Coupaye, T., *ASM: a code manipulation tool to implement adaptable systems*, *http://asm.ow2.org/current/asm-eng.pdf*, o.J.

9. Apache Foundation, *Apache Commons BCEL*, *https://commons.apache.org/proper/commons-bcel/manual.html*, 2014.

10. The Apache Software Foundation, *Class InstructionFactory*, *http://commons.apache.org/proper/commons-bcel/apidocs/org/apache/bcel/generic/InstructionFactory.html*, The Apache Software Foundation, 2014

11. Lindholm, T. and Yellin, F. and Bracha, G. and Buckley, A., *The Java® Virtual Machine Specification*, *Java SE 8*, Addison-Wesley 2014

12. o.A., *JDK 8*, *http://openjdk.java.net/projects/jdk8/*, OpenJDK, 2014

13. Plümicke, M., *Java Type System – Proposals for Java 10 or 11*, to appear in *Tagungsband zum 19. Kolloquiums Programmiersprachen und Grundlagen der Programmierung (KPS 2015)*, 2015

Applying Active Continuous Quality Control to Cross-Platfrom Conformance Testing

Johannes Neubauer[1] and Bernhard Steffen[1]

Technische Universität Dortmund, Germany
`johannes.neubauer|steffen@cs.tu-dortmund.de`

Abstract. In this paper we sketch how *Active Continuous Quality Control* (ACQC) can be adapted to *Learning-based Cross-platform Conformance Testing* (LCCT), i.e. to a method which is tailored to validate the preservation of behavioral equivalence after migration. Our method is designed to be applied e.g. after adaptations or technological switches in terms of operation system, programming language, execution environment, third-party components, optimizations, new access methods to the application, or API changes of third-party services. Technically, LCCT is based on a combination of our approach to higher-order integration of user-level test blocks with active automata learning in our learning framework *LearnLib*. Its impact has been shown by revealing migration-specific bugs typically stemming from browser-specific functionality.

Keywords: Active Learning, Model Checking, Testing, Migration, Validation, Model-Based Testing

1 Introduction

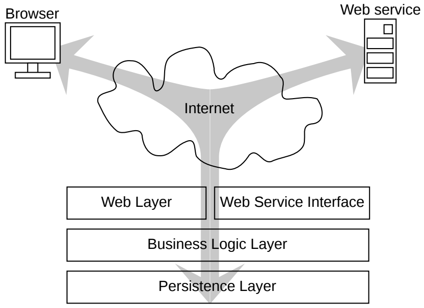


Fig. 1. The layered architecture of the enterprise application OCS

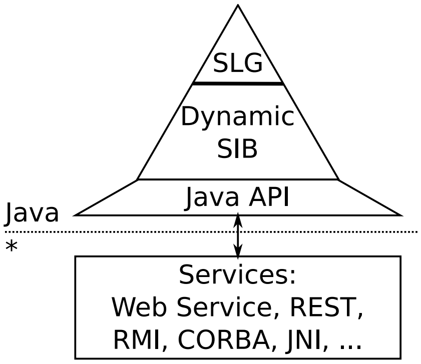


Fig. 2. The new dynamic SIB pattern of the jABC.

Modern software systems, in particular when successful, undergo continuous change and therefore require a continuous accompanying quality control at the system level. This is especially true for multi-layered component-based systems (see Fig. 1) in real-world scenarios, which can easily be changed by updating or exchanging components and their implementations [6]. Today's testing technology does not sufficiently address this problem as it requires enormous ongoing manual effort. E.g., keeping regression test suites or test models up to date is very costly, which is the major hurdle for model-based testing [30] to enter industrial practice. Recently, automata learning technology has been proposed to reduce the manual effort of the required test/model adaptation process. In [32] we presented *Active Continuous Quality Control* (ACQC), an approach that seamlessly integrates learning into the software evolution process in order to arrive at a truly continuous quality control. ACQC uses an incremental variant of active automata learning technology in order to closely monitor and steer the evolution of applications throughout their whole life-cycle with minimum manual effort. Key to this approach is the combination of the common practice of a periodic, e.g. daily, system build with a fully automatic testing process, performed and controlled via incremental active automata learning [24].

In this paper we sketch how ACQC can be adapted to *Learning-based Cross-platform Conformance Testing* [18] (LCCT), i.e. to a method which is tailored to validate the preservation of behavioral equivalence after migration [2]. Our method is designed to be applied e.g. after adaptations or technological switches in terms of operation system, programming language (e.g. reimplementation of legacy software), execution environment (e.g. application server), third-party components (e.g. database vendor, relational versus noSQL databases, or access-layer to the data-base), optimizations (e.g. caching or clustering), new access methods to the application (e.g. a RESTful API [4]), or API changes of third-party services (like Facebook, Twitter, or Google Maps). Technically, we combine our approach to higher-order integration of user-level test blocks with active automata learning in our learning framework *LearnLib* [14, 21]. Its impact has been shown by revealing migration-specific bugs typically stemming from browser-specific functionality.

In contrast to ACQC, where maintaining the stability of a common abstraction layer via a common learning alphabet during the evolution of the *System Under Learning* (SUL) as well as the cross-version reuse of information during the learning process was essential, these issues are not critical for LCCT. Rather, the third issue of ACQC, providing a correct mapping between the abstract level of learning and the concrete implementation, needs to be flexibilized for LCCT to deal with the platform specific versions of implementations. We employ the dynamic service integration feature of the jABC, our graphical application development framework [28] to elegantly solve this problem via higher-order test-block integration [19]. This allows us to realize LCCT elegantly as a learning process accompanied by a dynamic comparision of the platform-specific versions of the learned models. In essence, one could understand LCCT as an iteration between multiple learning phases which terminates when no differences are detected anymore:

- Hypothesis models for the various platform versions are learned independently.
- The differences between these hypothesis models are exploited to construct distinguishing traces which are used to trigger a new learning phase.

In this extended abstract we will focus on the mapping of abstract test symbols to concrete test-block implementations and its realization within the jABC, while we defer the presentation of the iterative LCCT process to the extended version of this paper. For a detailed elaboration of the technical details of dynamic service binding please refer to [18].

In the following, we will therefore first sketch the jABC including its pragmatics, before we address the higher-order test block generation in Sec. 3, followed by a short paragraph about model extrapolation in Sec. 4, a brief discussion of our example scenario in Sec. 5, and our conclusions in Sec. 6.

2 Extreme Model-Driven Design in the jABC

The user-level test blocks are realized in the jABC [13], a framework for service-oriented development that allows users to create services and applications easily by composing reusable building blocks into (flow-) graph structures that are both formally well-defined as well as easy to read and build. These building blocks are called *Service Independent building Blocks* (*SIBs*) in analogy to the telecommunication terminology [25], in the spirit of the service-oriented computing paradigm [11] and of the *one thing approach* [29], an evolution of the model-based lightweight coordination approach of [12] specifically applied to services.

On the basis of a large library of SIBs, the user builds models for the desired system in terms of hierarchical *Service Logic Graphs* [26] (*SLG*). SLGs are semantically interpreted as *Kripke Transition Systems* (*KTS*), a generalization of both *Kripke structures* (*KS*) and *labeled transition systems* [15] (*LTS*) that allows labels both on nodes and edges.

The service integration into the graphical process model design framework jABC is realized via dynamic service binding (see Fig. 2) that supports domain-specific (business) activities [3].

Dynamic service integration is technically achieved by directly binding a service in form of a Java method to an activity, denoted by a *dynamic SIB*. This is realized by considering services as *first-class objects* and therefore introducing a *type-safe second-order context* for exchanging services of a service graph at the parameter level, a step reminiscent of higher-order functions in functional programming languages [23]. The type-safe second-order context is a mapping from a tuple consisting of a name (e.g. ‘userController’) and type information (e.g. `UserController`) to a Java object (at runtime). Each entry in this mapping is called a *context variable*. Activities read and write values from context variables.

3 Higher-Order Test Block Integration

We apply the dynamic service integration [17] approach of the jABC framework to active automata learning. This provides us with the necessary flexibility to infer comparable models via automata learning for validating platform migrations.

Enabling dynamic service integration does not require any implementation of adapters for mapping of activities to services, as domain-specific activities are instead modeled hierarchically as SLGs themselves on the basis of low-level services:

- Services are provided as methods of a Java class or interface (i.e. a *remote enterprise bean* (EB)[1], a RESTful- or a Selenium service respectively), abstracting from technical details. These are integrated via dynamic SIBs in low-level graphs, which are absolutely unaware of the process models.
- A fully configured instance of a subclass of the service class or interface is read from the context as input to the corresponding dynamic SIB, and the configured method is executed[2] as the control-flow reaches the activity.
- Available SIB libraries in terms of low-level graphs allow application experts to build high-level, coarse-grained, and domain-specific test blocks.

Being organized in taxonomies, these SIB libraries can easily be discovered and (re)used for building complex process models, the aforementioned service logic graphs (SLG).

4 Test-Based Model Extrapolation

In our setting, active automata learning [24] may be interpreted as a systematic test generation framework that interrogates the SUL and extrapolates an appropriate corresponding (hypothesis) model. For learning reactive systems, like e.g., a web-services or, as in our example scenario, complete web applications, *Mealy machine* models have turned out to be the target model of choice. These can efficiently be learned using the LearnLib [20, 21, 14, 10], our framework for active automata learning. LearnLib[3] provides different active learning algorithms and optimization strategies for handling counterexamples, filter techniques that allow reducing the number of executed tests through domain knowledge [1, 8], further optimizations like parallelization [7], and, since recently [9], a space optimal version, which is also particularly well-suite to support the comparison phase of LCCT.

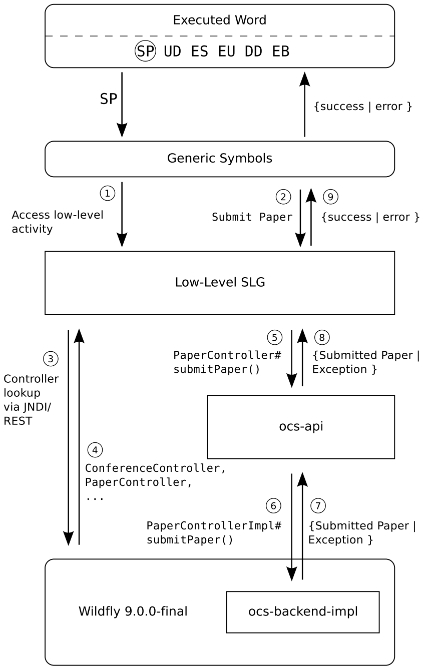

Executed Word
SP UD ES EU DD EB
SP
{success | error }
Generic Symbols
1
Access low-level activity
2
Submit Paper
9
{success | error }
Low-Level SLG
3
Controller lookup via JNDI/ REST
5
PaperController# submitPaper()
8
{Submitted Paper | Exception }
ocs-api
4
ConferenceController, PaperController, ...
6
PaperControllerImpl# submitPaper()
7
{Submitted Paper | Exception }
Wildfly 9.0.0-final
ocs-backend-impl

Fig. 3. The low-level SLGs dynamically accesses the remote EB or RESTful API of the OCS.

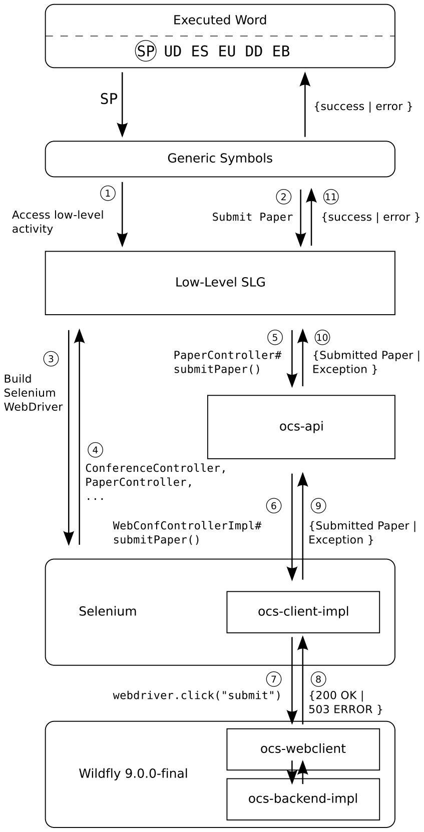


Fig. 4. The low-level SLGs dynamically accesses the web interface of the OCS via Selenium.

5 Example Scenario

The *Online Conference Service* (OCS) [16] is a multi-layered enterprise application (see Fig. 1) where, in particular, the presentation layer (*frontend*) is separated from the business logic (*backend*). The API of the backend is partitioned into controller interfaces for every type of entities modeling the domain of the OCS like a *conference*, a *paper*, or a *report*. A derived controller class implements the different actions that are possible on the respective objects, e.g., 'create a

[1] A Java EE technology to execute enterprise services remotely via *Remote Method Invocation* (RMI).

[2] We support both interpreted execution using the *Java Reflection API* [5] and full-code generation executing the method directly (i.e. type-safe) in the generated code.

[3] LearnLib is available at `http://www.learnlib.de`

user'. The used input symbols (cf. the edge labels in Fig. 5 in order to identify the abbreviations) are as follows:

`SP` *Submit Paper*: An author submits a paper to the conference but does not provide a document file. Since the number of papers in a conference is negligible for the overall workflow, we allow only one paper submission per conference.

`UD` *Upload Document*: An author uploads a document file for the previously submitted paper.

`DD` *Download Document*: The PC Chair downloads a document file of a paper.

`BD` *Bidding*: A PC Member submits a bidding for a paper.

`SA` *Special Assignment*: The PC Chair assigns a PC Member as reviewer iff the member has bided for the paper.

`SR` *Submit Report*: A reviewer submits a report for a paper.

`ES` *End Submission*: The PC Chair stops the submissions phase. From now on it is not possible to submit any new papers. This will also start the bidding phase and all PC Members will be able to submit biddings for papers.

`EU` *End Upload*: The PC Chair stops the upload phase. It is no longer possible to upload documents to papers.

`EB` *End Bidding*: The PC Chair stops the bidding phase. Members of the program committee are no longer allowed to bid for papers.

`EA` *End Assignment*: The PC Chair stops the assignment phase. From now on it is not possible to assign reviewers to papers. This will also start the review phase during which all reviewers are able to submit a report to an assigned paper.

The respective symbols will be successfully executed if all prerequisites are fulfilled. As in classical testing the membership queries have to be executed independently. In automata learning this is realized via a so-called *reset* that puts the SUL in a predefined state as all queries have to begin in the start state of the hypothesis automaton. In the case study the reset for every membership query creates a new conference and employs exactly one *PC chair*, *PC member*, and *author*. Since the number of papers in a conference is negligible for the overall workflow, we furthermore allow only one paper submission per test run.

In order to faithfully capture true web-based user behavior, we realized an alternative implementation of these controllers, denoted by *web-test controllers*, using the web-test framework Selenium [22]. These controllers, which truly mimic users operating the OCS via a browser, implement the same controller interface than their backend counterpart, and should ideally also have the same impact.

We have used our approach to dynamic service binding (cf. Sec. 3), in order to obtain variants of test blocks for the same API (controller interface) representing different implementations. This allowed us to dynamically exchange the implementation, i.e. remote EB or RESTful API (see Fig. 3) and the browser instrumentation (see Fig. 4) via higher-order service integration, and therefore to efficiently and elegantly coordinate the learning and comparison of the two platform-specific models.

Its impact has been shown by revealing migration-specific bugs typically stemming from browser-specific functionality as shown in Fig. 5. It shows an

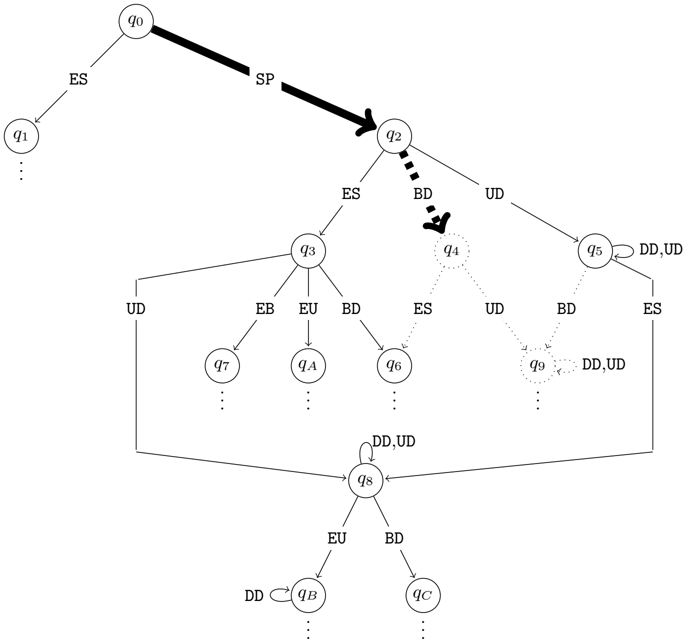


Fig. 5. Differences between learned models regarding the bidding [32].

excerpt of such an automaton of an OCS version with a bug in the security logic in the backend, which has been intercepted in the presentation layer. So it was not apparent by inspecting the presentation layer, only. For better readability we have omitted the output symbols (namely *success* and *error*) of the Mealy machine as well as the error edges (failed execution), since they are all reflexive. This is due to the rollback mechanism of the OCS, so that all erreneous executions have no impact on the state of the system.

The thick line followed by a thick dotted line in Fig. 5 shows that a member of the program committee may bid for reviewing a paper right after submission, although this should be possible after ending the submission phase (ES), only. This issue has also been found by our *active continuous control* (*ACQC*) approach [32]. However, in contrast to LCCT, ACQC tests the business logic only and therefore does not search for differences between frontend and backend, but it has been able to find the version which introduced the issue in the backend. Thus these are different test approaches that complement each other.

6 Conclusion and Future Work

With ACQC we have presented a novel approach to quality control that employs incremental active automata learning technology in order to periodically infer evolving behavioral automata of complex applications accompanying the development process. In this extended abstract we have presented the adaptation of ACQC to Learning-based Cross-platform Conformance Testing (LCCT), an approach specifically designed to validate successful system migration. Key to our approach is the combination of (1) adequate user-level system abstraction, (2) higher-order test-block integration, and (3) learning-based automatic model inference and comparison. LCCT employs second-order, type-safe execution semantics for (test) process models, which allows one to dynamically exchange the binding of functionality/test blocks at runtime. The impact of our approach has been illustrated along testing the conformance of the presentation layer and the business logic layer of Springer's Online Conference Service OCS in [18].

Complex multi-layered component-based systems can be subjected to rapid evolution, up to the point of exchanging components at runtime [31], a process which is called "online evolution". We are currently investigating how to extend our static approach, which focuses on managing changes that occur between system releases, to an approach capturing the effects of hierarchy [27] and self-adaptation. We think that it will be seen that our higher-order modeling approach is tailor-made for this purpose.

References

1. O. Bauer, J. Neubauer, B. Steffen, and F. Howar. Reusing system states by active learning algorithms. In *Eternal Systems*, volume 255 of *CCSE*, pages 61–78. Springer-Verlag, 2012.
2. A. De Lucia, R. Francese, G. Scanniello, and G. Tortora. Developing legacy system migration methods and tools for technology transfer. *Software: Practice and Experience*, 38(13):1333–1364, 2008.
3. M. Doedt and B. Steffen. An Evaluation of Service Integration Approaches of Business Process Management Systems. In *Software Engineering Workshop (SEW), 2012 35th IEEE*, 2012.
4. R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.
5. I. R. Forman and N. Forman. *Java Reflection in Action (In Action series)*. Manning Publications Co., Greenwich, CT, USA, 2004.
6. G. T. Heineman and W. T. Councill, editors. *Component-based software engineering: putting the pieces together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
7. F. Howar, O. Bauer, M. Merten, B. Steffen, and T. Margaria. The teachers' crowd: The impact of distributed oracles on active automata learning. In *ISoLA 2012*, Communications in Computer and Information Science, pages 232–247. Springer-Verlag, 2012.
8. H. Hungar, O. Niese, and B. Steffen. Domain-specific optimization in automata learning. In *Computer Aided Verification*, volume 2725 of *LNCS*, pages 315–327. Springer-Verlag, 2003.
9. M. Isberner, F. Howar, and B. Steffen. The ttt algorithm: A redundancy-free approach to active automata learning. In B. Bonakdarpour and S. Smolka, editors, *Runtime Verification*, volume 8734 of *Lecture Notes in Computer Science*, pages 307–322. Springer International Publishing, 2014.

10. M. Isberner, F. Howar, and B. Steffen. The open-source learnlib. In D. Kroening and C. S. Păsăreanu, editors, *Computer Aided Verification*, volume 9206 of *Lecture Notes in Computer Science*, pages 487–495. Springer International Publishing, 2015.
11. T. Margaria. Service is in the eyes of the beholder. *IEEE Computer*, 40:33–37, 2007.
12. T. Margaria and B. Steffen. Lightweight coarse-grained coordination: a scalable system-level approach. *STTT*, 5(2-3):107–123, 2004.
13. T. Margaria and B. Steffen. Agile it: Thinking in user-centric models. In *Leveraging Applications of Formal Methods, Verification and Validation, Proc. ISoLA 2008*, volume 17 of *Communications in Computer and Information Science*, pages 490–502. Springer Verlag, 2009.
14. M. Merten, B. Steffen, F. Howar, and T. Margaria. Next generation learnlib. In *TACAS 2011*, volume 6605 of *LNCS*, pages 220–223. Springer-Verlag, 2011.
15. M. Müller-Olm, D. Schmidt, and B. Steffen. Model-checking: A tutorial introduction. *SAS*, pages 330–354, 1999.
16. J. Neubauer, T. Margaria, and B. Steffen. Design for Verifiability: The OCS Case Study. In *Formal Methods for Industrial Critical Systems: A Survey of Applications*. John Wiley & Sons, 2011. In print.
17. J. Neubauer and B. Steffen. Second-order servification. In *ICSOB*, pages 13–25, 2013.
18. J. Neubauer and B. Steffen. Learning-based cross-platform conformance testing. 2015. In submission.
19. J. Neubauer, B. Steffen, and T. Margaria. Higher-order process modeling: Product-lining, variability modeling and beyond. *arXiv preprint arXiv:1309.5143*, 2013.
20. H. Raffelt, B. Steffen, and T. Berg. Learnlib: a library for automata learning and experimentation. In *FMICS '05*, pages 62–71. ACM, 2005.
21. H. Raffelt, B. Steffen, T. Berg, and T. Margaria. LearnLib: a framework for extrapolating behavioral models. *STTT*, 11(5):393–407, 2009.
22. Selenium. SeleniumHQ Web application testing system, visited july 2015. `http://seleniumhq.org/`.
23. P. Sestoft. Higher-order functions. In *Programming Language Concepts*, volume 50 of *Undergraduate Topics in Computer Science*, pages 77–91. Springer London, 2012.
24. B. Steffen, F. Howar, and M. Merten. Introduction to active automata learning from a practical perspective. In *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *LNCS*, pages 256–296. Springer-Verlag, 2011.
25. B. Steffen and T. Margaria. Metaframe in practice: Design of intelligent network services. In *Correct System Design, Recent Insight and Advances*, volume 1710 of *LNCS*, pages 390–415. Springer, 1999.
26. B. Steffen, T. Margaria, V. Braun, and N. Kalt. Hierarchical service definition. In *Annual Review of Communication*, pages 847–856. Int. Engineering Consortium Chicago (USA), IEC, 1997.
27. B. Steffen, T. Margaria, V. Braun, and N. Kalt. Hierarchical Service Definition. *Annual Review of Communications of the ACM*, 51:847–856, 1997.
28. B. Steffen, T. Margaria, R. Nagel, S. Jörges, and C. Kubczak. *Model-Driven Development with the jABC*, volume 4383 of *LNCS*, pages 92–108. Springer Berlin/Heidelberg, 2006.
29. B. S. T. Margaria. Business process modeling in the jabc: The one-thing approach. In *Handbook of Research on Business Process Modeling*, pages 1–26. IGI Global, 2009.

30. J. Tretmans. Model-Based Testing and Some Steps towards Test-Based Modelling. In M. Bernardo and V. Issarny, editors, *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *Lecture Notes in Computer Science*, pages 297–326. Springer-Verlag, 2011.
31. Q. Wang, J. Shen, X. Wang, and H. Mei. A component-based approach to online software evolution: Research articles. *J. Softw. Maint. Evol.*, 18(3):181–205, May 2006.
32. S. Windmüller, J. Neubauer, B. Steffen, F. Howar, and O. Bauer. Active continuous quality control. In *Proceedings of the 16th International ACM Sigsoft symposium on Component-based software engineering*, pages 111–120. ACM, 2013.

The Data-Flow Perspective on Static Single-Assignment Form

Baltasar Trancón Y Widemann and Markus Lepper

[1] TU Ilmenau
[2] semantics GmbH

Abstract. The static single-assignment (SSA) form is a low-level intermediate representation of computer programs. It has been designed to make the analysis and transformation of imperative code easier, by providing just the right amount of referential transparency of local variables. That SSA does its job fairly well is proven by its central use in many current state-of-the-art compilers. But the situation is something of a paradox. The languages that are translated to SSA typically possess many features that quite defy its purpose: Mutable arrays and objects with reference semantics on the one hand; concurrency and non-local control flow on the other. We take a fresh look at the design principles of SSA from the perspective of a domain-specific, semantically rigorous paradigm, namely total functional synchronous data-flow programming, embodied in the prototypic language Sig. We demonstrate that the expressivity of SSA is far more complete and foundational there. The same form has many interpretations, in notably as a data-flow diagram, the IR of a functional program, and both an intensional (Z schema-like) and a propositional definition of element-wise denotational semantics. We also demonstrate how the single operation particular to SSA, the phi node, naturally suggests semantically rigorous solutions to two principal semantical problems of the data-flow approach, namely initialization and control flow. Both are necessary for real-world applications, but notoriously ill-supported in many established practical programming systems.

Clean Java – Von Anfang an!

Anna Vasileva, Doris Schmedding

Technische Universität Dortmund, Deutschland
anna.vasileva | doris.schmedding@tu-dortmund.de

Zusammenfassung. In diesem Beitrag werden die Sensibilisierung der Entwickler für guten Code und die Integration von Qualitätsaspekten in einen Software-Entwicklungsprozess in der Informatik-Ausbildung vorgestellt. Für eine langfristige und erfolgreiche Verankerung des Themas Code-Qualität als Entwicklungsziel sind mehrere Vorgehensschritte notwendig. Neben den passend gewählten Werkzeugen zur statischen Codeanalyse, Metriken und Grenzwerte spielen eine hohe Motivation und Konzepte für die Beseitigung der gefundenen Mängel eine entscheidende Rolle für die Qualität des Codes und die Entwicklung eines besseren Programmierstils.

1 Einleitung

Das Software-Praktikum (SoPra) ist eine Bachelor-Veranstaltung, die in den ersten Semestern des Informatik-Studiums stattfindet. In der Realität sind die Studierenden, die diese Veranstaltung besuchen, zwischen 3. und 10. Semester. Im SoPra setzen die Studierenden in Software-Projekten die Inhalte der Software-Technik in die Praxis um. Erst im SoPra kommen die in den vorangehenden Veranstaltungen gelernten Techniken, Vorgehensmodelle und Konzepte der Programmiersprachen gemeinsam zum Einsatz. Die Studierenden setzen verschiedene Aspekte der Software-Entwicklung wie Planen, Modellieren mit UML, Programmieren und Testen um.

Die zufällig zusammengesetzten Achtergruppen bearbeiten gleichzeitig dieselben Aufgaben und werden dabei von Betreuern bzw. Betreuerinnen, die den Entwicklungsprozess gut kennen, unterstützt. Das Sopra wird dreimal im Jahr durchgeführt und es nehmen jeweils etwa 60-80 Studierende teil. Im einem SoPra werden jeweils zwei Projekte durchgeführt. Das erste Projekt ist meist eine Verwaltungsaufgabe. Im zweiten Projekt müssen die Studierenden oft ein Computerspiel realisieren. Das SoPra hat insgesamt einen Arbeitsumfang von etwa 180 Zeitstunden. Die Modellierung mit UML wird mit dem Tool Astah [1] durchgeführt. Zur Realisierung der Projekte wird die objektorientierte Programmiersprache Java eingesetzt. Die Programmierumgebung basiert auf Eclipse, in das einige nützliche Plugins integriert sind, wie z.B. Subclipse für den SVN-Zugriff und der WindowBuilder für die Gestaltung der grafischen Benutzungsschnittstelle.

Erst in der Zusammenarbeit im Team und der Umsetzung der Software-Entwicklungstechniken wird klar, wie wichtig die Lesbarkeit, die Verständlichkeit und die gute Wartbarkeit des entstehenden Programms sind. Trotzdem hat eine erste Messung (siehe Kapitel 3) mit Hilfe eines Werkzeugs zur statischen Code-Analyse gezeigt, dass für die Studierende die Qualität des Codes nicht im Fokus steht. Ein Grund dafür mag sein, dass in den vorangehenden Veranstaltungen der Schwerpunkt nicht auf

Code-Qualität liegt. Auch im SoPra versuchen die Studierenden in erster Linie für sie neue Technologien kennenzulernen und funktional korrekte Programme zu schreiben.

Die erste durchgeführte Analyse von studentischen Projekten [2] hat gezeigt, dass typische Mängel in folgenden Bereichen zu finden sind:

- Namensgebung und Java-Konventionen
- Komplexität und Länge der Methoden
- Verantwortlichkeit und Länge der Klassen

Mehr Details über die festgestellten Mängel sowie über die verwendete Metriken und Grenzwerte werden im Kapitel 2 vorgestellt.

1.1 Vorgehensweise

Um in der Lehrveranstaltung Software-Praktikum langfristig und nachhaltig für alle Jahrgänge von Studierenden die Erhöhung der Qualität des implementierten Codes zu gewährleisten, müssen in mehreren Iterationsschritten qualitätsverbessernde Maßnahmen in den Entwicklungsprozess integriert werden. Das nachfolgend vorgestellte Vorgehen orientiert sich an dem Plan-Do-Check-Act-Zyklus (PDCA) [3]. Gemäß des PDCA-Zyklus (siehe Abb. 1) wird eine Iteration geplant und durchgeführt. Danach werden die Ergebnisse überprüft und diskutiert. Anschließend folgt die nächste Iteration.

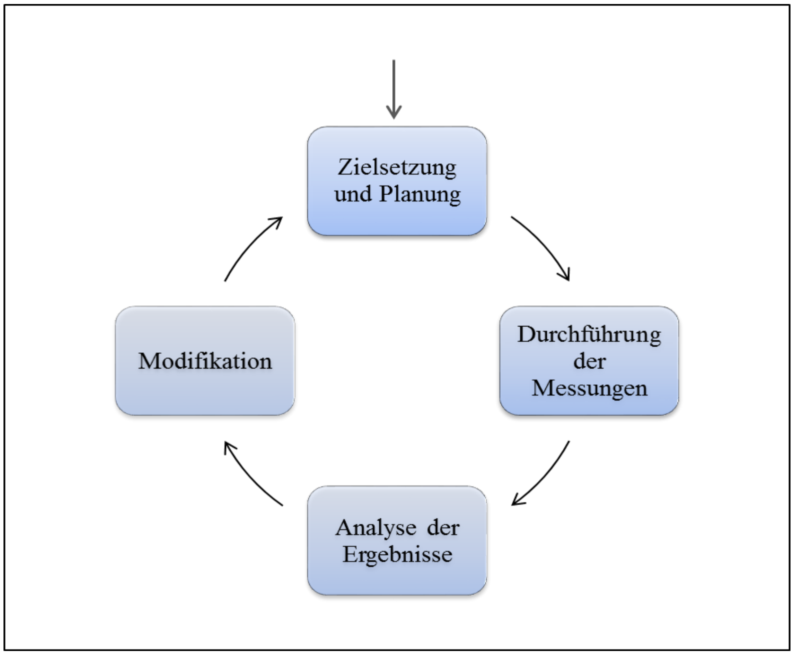


Abbildung 1: PDCA-Zyklus

In jedem Zyklus werden die Iterationsschritte wiederholt und die nächsten Ziele definiert. Wir beschreiben drei Zyklen, die zur Verbesserung der Codequalität durchgeführt wurden, bis erste gute Ergebnisse zu sehen waren.

Der Aufbau des Beitrags orientiert sich an dieser Vorgehensweise. In Kapitel 2 werden zunächst die Zielsetzung und Planung der ersten Iteration behandelt. Dann folgen in Kapitel 3 die Darstellung der ersten Messung, die Diskussion der Ergebnisse und die auf dieser Basis getroffenen Maßnahmen zur tieferen Verankerung des Themas Code-Qualität bei den studentischen Entwicklern. In der zweiten Iteration, die in Kapitel 4 vorgestellt wird, zeigen die getroffenen Maßnahmen leider keine positive Wirkung. Erst in der dritten Iteration, beschrieben in Kapitel 5, sind die neuen Maßnahmen erfolgreich. Es gelingt uns, das Thema Code-Qualität vom Anfang des Projekts an in den Fokus zu rücken. Der Beitrag schließt mit einem Fazit.

2 Zielsetzung und Planung

Gemäß der Goal-Question-Metrik-Methodik (GQM) [4] wurden als Erstes die Ziele für bessere Code-Qualität auf Basis typischer gefundener Mängel in studentischen Projekten definiert. Um diese Ziele zu erreichen, werden Fragen formuliert. Anschießend werden Metriken ausgewählt, mit deren Hilfe diese Fragen beantwortet werden können.

In seinem Buch *Clean Code* [5] präsentiert Robert Martin als erfahrener Software-Entwickler eine Fülle von typischen Qualitätsmängeln. Für unsere Veranstaltung wurden davon diejenigen ausgewählt, die häufig in studentischen Projekten vorkommen, die für die Studierenden leicht verständlich sind und die mit Tool-Unterstützung möglichst gut zu entdecken und zu beheben sind.

Für die Suche nach Mängeln in den studentischen Projekten wurde die Tool-unterstützte statische Code-Analyse angewendet. Diese können die Studierenden in der Implementierungsphase von Anfang an einsetzen, auch wenn der Code noch nicht kompilierbar ist. Tools zur statischen Code-Analyse bieten die Möglichkeit, gezielt nach verschieden Mängeln zu suchen und selbstdefinierte Grenzwerte zu verwenden. Selbstverständlich können die Messungen nicht völlig automatisiert durchgeführt werden. Der Code und insbesondere die Fundstellen müssen genau angeschaut werden, da solche Tools z.B. humorvolle oder sinnlose Bezeichner, die in studentischen Projekten häufig vorkommen, nicht finden können.

Das im SoPra verwendete Werkzeug zur statischen Code-Analyse ist PDM [6]. Aus den vielen zur Verfügung stehenden Tools zur statischen Code-Analyse, die sich teilweise sehr ähnlich sind, haben wir PMD gewählt, da PMD benutzerfreundlich und für Anfänger geeignet ist.

Auf Basis von typischen Defiziten in studentischen Projekten und den Ausbildungszielen im Bereich der Software-Qualität wurden für PMD eigene Regeln definiert [2]. Diese werden im XML-Format notiert und stehen allen GruppenbetreuerInnen sowie den Studierenden zur Verfügung. Wie bereits erwähnt, wurden die Ziele auf der Basis der Analyse der studentischen Projekte [2] und des Buches *Clean Code* [Mar09] festgelegt. Die gewählten Metriken und Grenzwerte orientieren sich am Projektumfang, den typischen Mängeln in studentischen Programmen sowie den in die Literatur [5, 6, 7, 8] vorgestellten Grenzwerten. In der

Literatur sind auch Statistik-basierte Grenzwerte zu finden, die z. B. nicht einfach übernommen werden können, da der in studentischen Projekten geringere Umfang und die geringere Erfahrung in Betracht gezogen werden müssen.

Die ausgewählten Qualitätsaspekte werden in die drei Gruppen Bezeichner, Methoden und Klassen eingeteilt. Angegeben sind jeweils außerdem die zur Messung verwendeten Metriken und die gewählten Grenzwerte.

2.1 Bezeichner

Eine gute Bezeichnerwahl ist von großer Bedeutung für die Lesbarkeit und die Verständlichkeit des Programmcodes. Die Analyse der studentischen Projekte hat gezeigt, dass die Studierende dazu neigen, kurze, wenig sinnvolle oder humorvolle Bezeichner zu wählen, was den Regeln für hohe Code-Qualität widerspricht.

Die Entdeckung der schlechten Bezeichner erfolgt mit Hilfe von PMD. Das Werkzeug prüft die Länge der Bezeichner. Da erfahrungsgemäß kurze Bezeichner inhaltlich nicht aussagekräftig sind, ließen wir die vier oder weniger Zeichen langen Bezeichner vom Tool herausfiltern.

Die Bezeichner müssen auch manuell, stichprobenartig kontrolliert werden. Auch wenn die Bezeichner gemäß der Regeln für die Länge gewählt wurden und die Java-Konventionen eingehalten werden, können sie dennoch sinnlos oder humorvoll sein, oder missverständliche Information liefern.

Auch die Einhaltung der Java-Konventionen zur Namensgebung kann von Werkzeugen zur statischen Code-Analyse größtenteils kontrolliert werden. PMD kann nicht erkennen, ob der Bezeichner einer Methode mit einem Verb anfängt, PMD kann aber z.B. feststellen, ob ein Klassenbezeichner mit einem Großbuchstaben beginnt.

2.2 Methoden

Die Länge der Methoden und die zyklomatische Komplexität werden ebenso mit dem Werkzeug zur statischen Code-Analyse untersucht. Aufgrund der von uns definierten Regeln erkennt PMD Methoden als mangelhaft, die länger als 40 LOC (Lines of Code) sind, wobei die Kommentare und die leere Zeilen mitgezählt werden. Nach Martin [5] dürfen die Methoden nicht länger als vier Zeilen sein, wobei jede Schleife nur eine Zeile Code beinhalten darf. McConnell [7] sagt, dass eine Methode zwischen 100 und 200 Zeilen haben darf. Der gewählte Grenzwert von 40 LOC ist für den relativ geringeren Umfang der Projekte und noch unerfahrene Entwickler akzeptabel und führt zu Methoden, die überschaubar sind, so dass Fehler schnell und einfach gefunden werden können.

Die Komplexität von Methoden als wichtiger Aspekt der Code-Qualität wird häufig in Form der zyklomatischen Komplexität gemessen. Diese lässt sich mit Hilfe von Tools zur statischen Code-Analyse leicht kontrollieren. Für die Analyse der studentischen Projekte wurde der Grenzwert von 10 von PMD übernommen [6]. Dieser darf nicht überschritten werden.

Neben der zyklomatischen Komplexität und der Länge der Methoden wurde die Anzahl der Parameter untersucht. Wenn die Anzahl der Parameter einer Methode vier überschreitet, dann meldet PMD nach den von uns definierten Regeln die Stelle als mangelhaft. Martin [5] sieht eine maximale Länge von drei vor, wobei drei Parameter nur in Ausnahmefällen akzeptabel seien. PMD sieht in den mitgelieferten Regeln einen Maximalwert von 10 vor.

Eine Parameterliste darf nicht zu lang sein, weil diese schwer zu verstehen und zu benutzen wird. Außerdem ändern sich solche Listen beim Implementieren ständig. Die Nutzung vieler Parameter kann dazu führen, dass die Parameter des gleichen Typs verwechselt werden können. Lange Parameterlisten können auch zur Entstehung von redundantem Code führen. [9]

2.3 Klassen

Ebenso wie lange Methoden müssen auch lange Klassen vermieden werden, um eine hohe Übersichtlichkeit und eine gute Testbarkeit und Wiederverwendbarkeit zu erreichen. Die Länge der Klassen sowie die Anzahl der Klassen hängt laut [8] vom Projektumfang ab. Da der Projektumfang im SoPra nicht so hoch ist, haben wir in den SoPra-Regeln festgelegt, dass PMD eine Klasse als zu lang erkennen soll, wenn diese mehr als 400 LOC hat. Standardmäßig ist hierfür bei PDM 1000 Zeilen eingestellt [6].

Bei der Messung soll auch geprüft werden, ob eine Klasse zu viel Verantwortung übernimmt, man spricht dann von einer Gott-Klasse. Die Definition von Gott-Klassen und die Kriterien, die die Entdeckung unterstützen, wurden von Lanza und Marinescu [8] übernommen.

PMD erkennt eine Gott-Klasse, wenn alle folgenden Kriterien verletzt sind:

- Die Summe der zyklomatischen Komplexität aller Methoden einer Klasse (WMC - Weighted Method Count) darf den Grenzwert von 47 nicht überschreiten.
- Die Anzahl der direkten Zugriffe einer Klasse auf die Attribute anderer Klassen (ATFD – Access To Foreign Data) darf nicht höher als 5 sein.
- Die dritte Metrik (TCC - Tight Class Cohesion) misst die Rate der direkt gekoppelten public-Methoden einer Klasse geteilt durch die maximale Anzahl der Verbindungen der Methoden. Dieser Wert sollte 0,33 nicht unterschreiten. Nur wenn er höher als 0,33 ist, wird nach der Definition von Lanza und Marinescu [8] die gewünschte Kohäsion der Methoden gewährleistet.

Zwei Methoden sind als verbunden anzusehen, wenn sie auf die gleichen Instanzvariablen einer Klasse zugreifen. Innerhalb einer Klasse sollten die Methoden eine hohe Kohäsion besitzen. Andernfalls kann man die Methoden leicht auf zwei Klassen aufteilen.

3 Die erste Messung

Die erste Messung fand im Ferien-SoPra im SS 14 statt. Die Studierenden wussten beim Programmieren nicht, dass der Programmcode untersucht wurde. Nach der Messung am Ende des SoPras wurden die festgestellten Mängel den Gruppen detailliert präsentiert. Die Gesamtergebnisse sind in Abbildung 2 dargestellt.

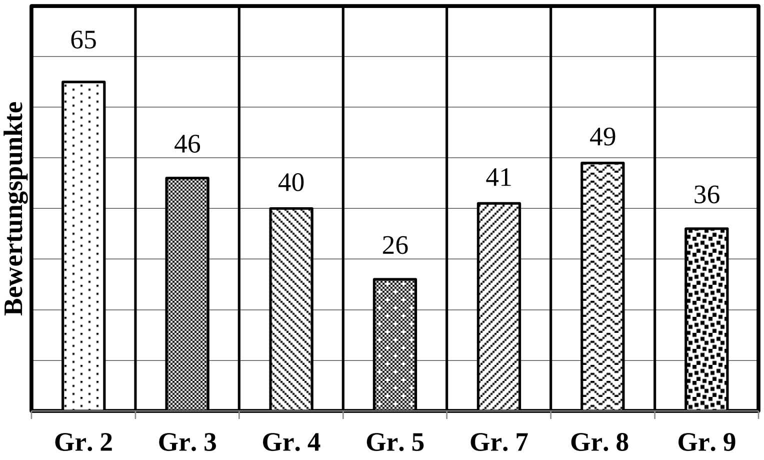


Abbildung 2: Gesamtergebnis im Freien-SoPra im Sommer 2014

Zu sehen sind die von den Gruppen jeweils insgesamt erreichten Punkte. Jede Gruppe startet mit einem Pluspunkte-Konto von 95 Punkten, aufgeteilt in die oben definierten drei Kategorien (Bezeichner, Methoden und Klassen) und „Sonstiges“. Bei jedem gefundenen Verstoß wird für die entsprechende Kategorie ein Punkt abgezogen.

In Bezug auf den Qualitätsaspekt Bezeichnerwahl haben wir folgendes festgestellt. Wie bereits erwähnt, misst das Tool PMD die Länge der Bezeichner, über deren Aussagekraft das Tool keine Entscheidung treffen kann. Deshalb müssen die Fundstellen manuell geprüft werden. Wir haben Bezeichner gefunden, die zwar kurz, im Aufgabenkontext aber sinnvoll sind, wie z.B. “Zug” oder “Game” als Klassenbezeichner. Diese werden nicht als Mängel betrachtet. Bei anderen Bezeichnern der Art “m1”, “m2” oder “p” fehlt die Aussagekraft, und sogar mit gutem Kontextwissen sind sie schlecht zu verstehen. Derartige Bezeichner werden gerne für Methodenparameter und lokale Variable verwendet.

Die erste Messung ergab außerdem, dass die Java-Namenskonventionen von den Studierenden weitgehend eingehalten wurden. In der Kategorie Namensgebung blieben deshalb trotz oft schlechter Bezeichnerwahl einige Punkte erhalten.

Auffällig war, dass einige Gruppen nur zwei bis drei Methoden mit zu hoher zyklomatischer Komplexität in ihrem Projekt hatten, wohingegen andere Gruppen sehr viele derartige Methoden entwickelt haben. Eine Gruppe hatte eine Methode mit einer zyklomatischen Komplexität von 67 geschrieben.

Die Methoden, die von PMD als zu komplex erkannt wurden, waren auch diejenigen, die offensichtlich zu lang waren. Im Programmcode der Gruppe 9 wurde vom PMD eine Methode mit 186 Zeilen und einer zyklomatischen Komplexität von 49 entdeckt.

Lange Klassen sind im SoPra eher selten, da der Projektumfang nicht so groß ist. Die erste Messung hat gezeigt, dass viele Klassen, die von PMD als zu lang erkannt wurden, auch als Gott-Klassen identifiziert wurden. Die längste Klasse hatte 992 LOC. Diese Klasse hat auch die Gott-Klasse-Regel von Lanza und Marinescu [8], die in Kapitel 2.3 vorgesellt ist, verletzt. WMC betrug 126, beim Grenzwert von 47, ATFD war fast 10-fach größer als der Grenzwert und TCC erreichte etwa 0.042. In dieser Klasse wurden auch auskommentierter und unbenutzter Code gefunden, wie z.B. Parameter, Attribute sogar ganze Methoden. Diese „Baustellen“ wurden auch als Mängel notiert.

Zum Punktabzug haben auch die schlechte Sortierung der Komponenten in den Klassen sowie tief verschachtelte if-Anweisungen geführt.

3.1 Analyse der Ergebnisse

Im Rahmen der ersten Messung wurden viele der bei der Planung angenommenen Mängel [10] festgestellt, was für die passende Wahl der betrachteten Qualitätsaspekte, der Metriken und des Code-Analyse-Werkzeugs spricht. Da nicht nur die Auswahl der Metriken, sondern auch die der passenden Grenzwerte sehr wichtig für die Qualitätsmessung und die Erreichbarkeit der Ziele sind, liefert die Evaluation auch Information darüber, ob die Grenzwerte gut gewählt und erreichbar sind. Die Erreichbarkeit der Ziele ist eine Voraussetzung für die Akzeptanz der Messung, die Erhöhung der Motivation und zur Verbesserung des Programmierstils der Entwickler.

Ein Grund für die weitgehende Einhaltung der Java-Konventionen kann die Zusammenarbeit im Team beim Erstellen der UML-Modelle sein, aus denen die Java-Code-Rahmen generiert werden. Weiterhin stellten wir besonders viele einbuchstabige Bezeichner bei den Parametern in Programmteilen fest, die von Einzelpersonen erstellt wurden. Diese Programmteile wurden bis dahin keinem definierten Code-Review-Prozess unterzogen.

Nach unseren Erfahrungen in studentischen Projekten sind die langen Methoden auch diejenigen, die eine hohe zyklomatische Komplexität aufweisen.

Alle festgelegten Grenzwerte haben sich als geeignet erwiesen, da sie von Studierenden eingehalten werden konnten. Auch die ausgewählten Metriken haben sich aus unserer Sicht für das Ziel bewährt, die Lesbarkeit und Verständlichkeit des Codes zu steigern, da sie den Studierenden überwiegend gut vermittelbar waren.

3.2 Modifikationen

Die Ergebnisse der Messungen und damit die erreichte Code-Qualität wurden allerdings nicht als zufriedenstellend angesehen. Folgende Maßnahmen wurden deshalb getroffen:

- Das Thema Code-Qualität wird von Anfang an in den Ablauf den SoPras integriert.
- Die Messung wird in der Einführungsveranstaltung angekündigt.
- Zur weiteren Verankerung des Themas Code-Qualität wurden Diskussionen mit den Studierenden über ihre Ergebnisse nach dem ersten Projekt geplant.
- Außerdem wird als flankierende Maßnahme stärker auf das Thema Refactoring eingegangen. Den Studierenden wird in einer Präsentation gezeigt, wie man die in Eclipse vorhandenen Refactoring-Techniken einsetzt.
- Zusätzlich wird den Studierenden im SoPra-Wiki [11] Lernmaterial in Form von Tutorials zu den Themen PMD, Clean Code und Refactoring zur Verfügung gestellt.

Unter Refactoring [9] versteht man Techniken, die zur Verbesserung des Quellcodes zu verwenden sind, wobei aber die Funktionalität bestehen bleibt. Da man nicht davon ausgehen kann, dass studentischer Programmcode absolut mängelfrei ist, werden Refactoring-Techniken zu jedem von uns als besonders relevant betrachteten Qualitätsbereich eingeführt.

3.2.1 Refactoring für Bezeichner

Die Qualitätsaspekte Namensgebung und Java-Konventionen sind für die Studierende sehr einfach zu realisieren, weil diese gut nachvollziehbar und Mängel leicht zu beseitigen sind. Mit Hilfe der Refactoring-Technik *Rename* lassen sich alle Bezeichner schnell und ohne großen Aufwand verbessern. Ein Beispiel dafür ist, dass ein Mitglied der Siegergruppe 2 (s. Abb. 2) alle kurzen Bezeichner mit Hilfe von *Rename* vor der Messung ersetzt hat. Deswegen hat diese Gruppe in diesem Bereich die volle Punktzahl, wogegen andere Gruppen keinen Punkt behalten konnten.

3.2.2 Refactoring für Methoden

Die Lesbarkeit, die Komplexität und die Länge der Methoden können mit den Refactoring-Techniken *Extract Method* oder *Extract Local Valiable* verbessert werden. Durch *Extract Lokal Variable* wird ein Ausdruck durch eine neue lokale Variable ersetzt. Durch *Extract Method* wird ein Block von Anweisungen einer Methode in eine neue Methode ausgelagert. Diese Technik kann auch eingesetzt werden, um die Verständlichkeit des Codes zu erhöhen, z.B. wenn ein Teil der Methode gut kommentiert werden muss, damit die Funktionalität besser verstanden wird. [9]

Allerdings ist der Aspekt Refactoring für Methoden nicht so einfach umzusetzen. Ob die Funktionalität erhalten geblieben ist, muss deshalb durch Tests überprüft werden. Durch *Extract Method* entsteht ein neuer Sichtbarkeitsbereich für Instanz-Variable und Parameter. Wenn beispielsweise mehrere lokale Variable in dem Bereich, der zum Extrahieren markiert ist, deklariert und ihnen Werte zugewiesen werden, ist *Extract Method* nicht anwendbar.

Die Anzahl der Parameter lässt sich mit Hilfe von *Introduce Parameter Object* reduzieren. Auf dieser Weise wird eine neue Klasse erzeugt und mehrere Parameter einer Methode lassen sich durch ein Objekt dieser Klasse ersetzen. So wird aber nicht unbedingt eine bessere Lesbarkeit und Übersichtlichkeit gewährleistet, da durch das Erzeugen von neuen Klassen die gesamte Struktur geändert wird.

In studentischen Projekten sind auch Literale zu finden, meist in Form von konkreten Zahlenwerten in Bedingungen. Diese sind bei der Fehlersuche oder Wartung schwer zu verstehen, selbst wenn Kontextwissen vorhanden ist. Mit Hilfe von *Extract Constant*-Refactoring können die Zahlen einfach durch eine statische finale Variable mit aussagekräftigem Bezeichner ersetzt werden. Dadurch wird der Code verständlicher und auch wartungsfreundlicher.

3.2.3 Refactoring für Klassen

Die Beseitigung einer Gott-Klasse oder einer zu langen Klasse kann erreicht werden, indem durch das Verschieben von Methoden ein geeigneter Teil des Codes in eine andere Klasse verschoben wird. Zur Reduzierung der Länge und der Komplexität der Klasse können *Extract Class* oder *Move* der richtige Weg sein. Das Aufspalten einer zu langen Klasse in zwei kann durch *Extract Class* erreicht werden. Das ist leider oft mit großem Aufwand verbunden, deshalb muss man sich fragen, ob sich der Aufwand lohnt [9].

Für die Sortierung der Komponenten in einer Klasse und die Beseitigung von unbenutztem Code gibt es keine speziellen Refactoring-Techniken. Diese Mängel müssen manuell behoben werden. *Extract Method* kann verwendet werden, um der redundanten Code zu beseitigen.

4 Die zweite Messung

Im zweiten Einsatz zur Integration der Qualitätsaspekte in den Entwicklungsprozess im WS 14/15 wurde das Thema „Clean Code“ eingeführt, die Messung von Anfang an angekündigt und erläutert, nach welchen Mängeln bei der Analyse gesucht wird und wieso. Tutorials zur statischen Code-Analyse und Refactoring wurden bereitgestellt.

4.1 Analyse der Ergebnisse

Die Ergebnisse der Messung sind im Vergleich zu denen im SS14 trotz aller getroffenen Maßnahmen immer noch nicht besser. Bei der Präsentation und Diskussion der Ergebnisse des ersten Projekts zeigten sich die Studierenden durchaus einsichtig. Dennoch waren auch die Ergebnisse des zweiten Projekts im WS 14/15, die in Abbildung 3 dargestellt sind, nicht besser als die im ersten Projekt und die des vorangehenden Software-Praktikums im ersten Zyklus (siehe Abb. 2).

Die knappe Zeit im Projekt wird offenbar lieber in das schöne Aussehen der Benutzungsschnittstelle, in komplexe Algorithmen und noch mehr Funktionalität als in statische Code-Analyse und Refactoring investiert. Weiterhin scheint bei den studentischen EntwicklerInnen das Ziel, eine hohe Code-Qualität zu erreichen, eine deutlich geringere Priorität als ein schönes User-Interface und die Funktionalität des Programms zu besitzen.

Es reicht offenbar nicht aus, auf die Einsicht der Studierenden zu vertrauen. Obwohl die Betreuer bereits für das Thema sensibilisiert waren, konnten sie bei den Gruppen die tägliche Code-Analyse und mehr Sorgfalt im zweiten Projekt nicht erreichen. Die Studierenden haben zwar eine Code-Analyse am Ende des Projekts durchgeführt, als es kaum noch Zeit mehr gab und viele Mängel sich nicht mehr so leicht, z. B. mit Hilfe von einfachen Refactorings, beseitigen ließen, so dass viele Mängel erhalten blieben.

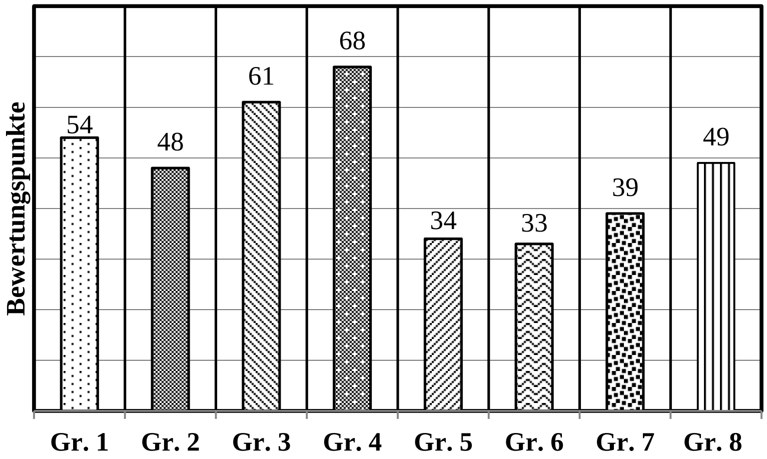


Abbildung 3: Gesamtergebnis im WS-SoPra 14/15

4.2 Modifikation

Die GruppenbetreuerInnen sollen frühzeitig mit den Gruppen zusammen Messungen der Code-Qualität durchführen und die Studierenden auf potenzielle Fehler und Mängel aufmerksam machen. Sie sollen mit den Gruppen diskutieren, wie man potentielle Mängel frühzeitig erkennen kann oder wie sich diese mit Hilfe von Refactorings beseitigen lassen, um die Motivation der Studierenden zu erhöhen.

Da es offenbar nicht ausreicht, über die Möglichkeit von Qualitätsmessungen und Maßnahmen zur Qualitätsverbesserung informiert zu sein, muss ein anderer Weg beschritten werden. Die Studierenden müssen "gezwungen" werden, die vorhandenen Tools und Techniken selbst auszuprobieren und anzuwenden.

5 Die dritte Messung

In der Einführungsveranstaltung wird das Thema Code-Qualität angesprochen. Die Studierenden führen eigenständig Messungen mit PMD und dem SoPra-Regelsatz durch. Die GruppenbetreuerInnen weisen wiederholt auf die Relevanz der Code-Qualität für das Projekt hin.

Nach dem ersten Projekt folgte eine Diskussion der Ergebnisse der durchgeführten Messungen mit den Gruppen und mit den BetreuerInnen. Vorschläge zur Vermeidung der festgestellten Mängel werden gesammelt.

Die Studierenden werden nach dem ersten und vor dem zweiten Projekt aufgefordert, die gefundenen Mängel mit Hilfe von Refactoring-Techniken zu beheben und einen Bericht darüber zu verfassen. Wenn das Verwenden von Refactoring nicht erfolgreich war, mussten die Studierenden dies schriftlich begründen. Viele Mängel insbesondere im Bereich der Namensgebung ließen sich problemlos z.B. durch das

Refactoring *Rename* beheben. Um die Länge der Methoden zu verkürzen und ihre Komplexität zu verringern konnte das Refactoring *Extract Method* oft erfolgreich eingesetzt werden. In manchen Fällen waren die Studierenden trotz ihrer Bemühungen zur Beseitigung der Mängel nicht in der Lage, dann konnten sie gut erklären, woran sie gescheitert waren.

Insbesondere zeigte sich in den zugesandten Berichten, dass die Studierenden trotz offensichtlicher Anstrengungen damit überfordert waren, die Gottklassen im Nachhinein zu beseitigen. Lange Parameterlisten wurden wie vorgesehen durch *Introduce Parameter Object* beseitigt. Die neu entstandene Struktur des Programmsystems wurde allerdings auch von den Studierenden kritisiert.

Insgesamt zeigte sich, dass die Mängel teilweise nur schwer behebbar waren und sich trotz Einsatz mehrerer Refactorings im Nachhinein nicht mit vertretbarem Aufwand beseitigen ließen.

Diese praktische eigene Erfahrung der Studierenden und der Einsatz dieser didaktischen Methoden haben dazu geführt, dass sich die Entwicklerteams bereits in der Modellierungsphase des zweiten Projektes Gedanken über eine gute Bezeichnerwahl und die Vermeidung von langen Methoden, langen Parameterlisten und Gott-Klassen gemacht haben. Bei dem Implementieren haben einige Gruppen versucht, der Regel vom Martin [5] zu folgen, dass eine Methode nur vier Zeilen lang sein sollte.

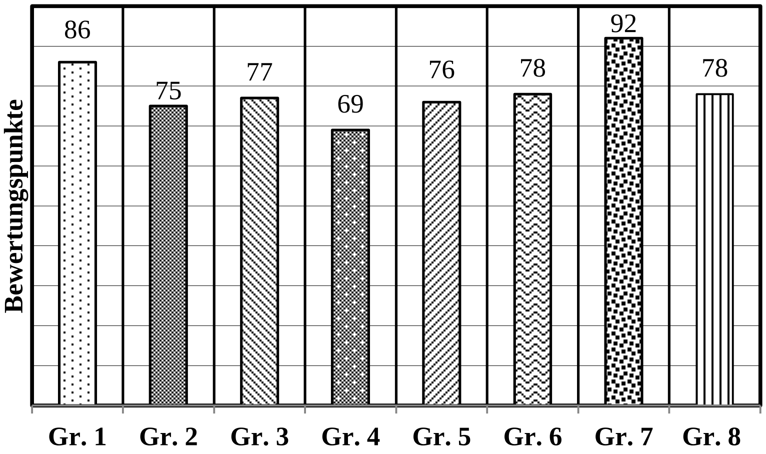


Abbildung 4: Gesamtergebnis im Freien-SoPra im Winter 2015

5.1 Analyse der Ergebnisse

Abbildung 4 stellt das Gesamtergebnis des zweiten Projekts im Ferien-SoPra nach WS 14/15 dar. Ein Vergleich der Werte in Abbildung 1 bis 4 zeigt, dass die besten Ergebnisse bei der vorerst letzten Iteration erzielt wurden. Besonders auffallend ist, dass die Ergebnisse aller Gruppen sehr hoch sind (zwischen 69 und 92).

Hervorzuheben ist, dass sich lange Parameterlisten nur noch bei Gruppe 4 finden. Die Hälfte der Gruppen konnte Methoden mit erhöhter zyklomatischer Komplexität vermeiden.

Aber fast alle Gruppen hatten immer noch Gott-Klassen. Das waren in der Regel die Klassen, die die Strategie der Spiele realisieren.

Mängel in den anderen Bereiche, die in Kapitel 2 vorgestellt worden sind, waren kaum noch zu finden.

5.2 Modifizieren

Die Auswertung der letzten durchgeführten Messung hat gezeigt, dass die Ergebnisse besser geworden sind, so dass wir von einer Verankerung des Themas Code-Qualität von Anfang der Entwicklung an durch die eingeführten didaktischen Maßnahmen ausgehen können. Dennoch sind noch einige Veränderungen in Bezug auf die Metriken notwendig.

Da die absoluten Methoden- und Klassenlängen gemessen wurden, gab es kritische Anmerkungen und Diskussionen zu den Messergebnissen. Dementsprechend besteht die Gefahr von adaptivem Verhalten, wie z. B. die Reduzierung der Kommentare. Um das zu vermeiden, wird bei der zukünftigen Messungen die Metrik NCSS (Non Commenting Source Statements) verwenden. Diese lässt Kommentare und die leere Zeilen unberücksichtigt.

Die Definition der Gott-Klassen ist für die Studierenden schwer zu verstehen und einmal entstandene Gott-Klassen sind durch Refactoring schwer zu beseitigen. Deshalb ist es notwendig, dass zu lange Klassen und zu komplexe Methoden bereits bei der Modellierung vermieden werden. Oft lässt sich schon bei der Modellierung erkennen, dass eine Klasse oder eine Methode zu viel Verantwortung übernimmt. Finden sich mehrere dieser Methoden in einer Klasse, so ist die Gefahr einer Gott-Klasse sehr groß.

6 Fazit

Eine langfristige Integration von Qualitätsaspekten konnte durch die Anwendung von didaktischen Maßnahmen in mehreren Iterationszyklen gewährleistet werden. Ein wichtiges Ergebnis ist, dass sich in einem modellbasierten Entwicklungsprozess eine hohe Code-Qualität mit vertretbarem Aufwand nur erreichen lässt, wenn dieses Ziel von den EntwicklerInnen bereits während der Modellierungsphase beachtet wird. Die Beseitigung von Mängeln am Ende der Implementierungsphase, wenn ein Tool zur statischen Code-Analyse auf die Mängel aufmerksam macht, ist nicht wirklich zielführend. Am Ende des Projekts ist einerseits die Zeit immer knapp und andererseits hat sich gezeigt, dass die Behebung mancher Mängel sehr aufwendig ist und Programmieranfänger überfordert. Komplexe Änderungen müssen durch die erneute Durchführung der Funktionstests überprüft werden. Komplexe Refactorings verändern die Struktur des Systems und können weitere Refactorings notwendig machen. So entstehen Kettenreaktionen, denen Programmieranfänger nicht gewachsen sind, und die Motivation der Studierenden leidet darunter.

Code-Qualität lässt sich also nur erreichen, wenn dieses Ziel von Anfang an verfolgt wird.

Literaturverzeichnis

[1] Astah, *http://astah.net/,* abgerufen am 09.07.2015.

[2] J. Remmers, *Code-Qualität im Software-Praktikum,* Bachelorarbeit, Fakultät für Informatik, TU Dortmund., 2014.

[3] A. Syska, *Produktionsmanagement,* Gabler, 2006.

[4] V. R. Basili, G. Caldiera und H. D. Rombach, *The Goal Question Metric Paradigm,* In: Encyclopedia of Software Engineering - 2 Volume Set, John Wiley & Sons, 1994.

[5] R. C. Martin, Clean Code, Prentice Hall, 2009.

[6] PMD, *http://pmd.sourceforge.net/,* abgerufen am 09.07.2015.

[7] S. McConnell, Code Complete, Unterschleißheim: Microsoft Press, 2007.

[8] M. Lanza und R. Marinescu, Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the design of Object-Oriented Systems, Springer, 2006.

[9] M. Fowler, *Refactoring - Wie Sie das Design vorhandener Software verbessern,* Köln: Addison-Wesley, 2000.

[10] D. Schmedding, A. Vasileva und J. Remmers, *Clean Code - ein neues Ziel im Software-Praktikum,* Dresden: SEUH, 2015.

[11] SoPra, *https://sopra.cs.tu-dortmund.de/wiki/start,* abgerufen am 09.07.2015.

[12] Healt4J, *http://www.main-gruppe.de/cms/opencms,* abgerufen am 10.06.2015.

[13] Checkstyle, *http://checkstyle.sourceforge.net/,* abgerufen am 08.06.2015.

[14] N. Fenton und J. Bieman, Software Metrics: A Rigorous and Practical Approach, 2015: CRC Press.

[15] A. J. Riel, *Object-oriented design heuristics,* 1996: Addison-Wesley Publishing.

[16] D. Schmedding, *Ein Prozessmodell für das Software-Praktikum,* Zürich: SEUH, 2001.

[17] A. Syska, *Produktionsmanagement,* Gabler, 2006.

Automatisierte Bewertung und Erzeugung von Übungsaufgaben zu Prinzipien von Programmiersprachen

Johannes Waldmann

Fakultät IMN, HTWK Leipzig,
johannes.waldmann@htwk-leipzig.de

Zusammenfassung Das E-Learning/E-Assessment-System autotool ist eine Online-Plattform zur semantischen Bewertung von studentischen Lösungen von Übungsaufgaben und gestattet auch das zufällige Generieren von Aufgabeninstanzen.
Ich verwende diese Plattform für einen Teil des Übungsbetriebes zur Vorlesung *Prinzipien von Programmiersprachen* und berichte über Design, Implementierung und Erfahrung mit Aufgabentypen zur polymorphen Typisierung, zur denotationalen Semantik rekursiver Definitionen, zur (approximierten) Spur-Semantik imperativer Programme, sowie zu dynamischen und statischen Ketten bei lokalen Unterprogrammen.

1 Einleitung

In der Vorlesung *Prinzipien von Programmiersprachen* geht es um Syntax, Semantik und Pragmatik programmiersprachlicher Konzepte und ihrer Realisierungen. Welche Übungsaufgaben sollte man dazu stellen?

Naheliegend sind Bastelarbeiten in verschiedensten realen Programmiersprachen. Oft sind dabei aber die jeweils betrachteten Konzepte nicht in Reinform vorhanden und die Studenten werden durch nebensächliche Sprachspezifika abgelenkt. Die reale Sprachvielfalt täuscht auch, denn unter der Oberfläche findet man doch oft nur das imperative objektorientierte Paradigma, und die Betrachtung der $(n+1)$-ten Instanz davon bringt dann nicht mehr viel neues.

Nützlich sind vielmehr Aufgaben zur isolierten Behandlung jeweils einer Idee. Ich realisiere solche Aufgaben als Semantikmodule für mein E-Learning/E-Assessment-System *autotool* [RW02]. Für jeden Aufgabetyp definiert dabei man eine problemspezifische Sprache. Dabei ist die Syntax entweder uniform (und entspricht der Syntax von Daten-Termen in Haskell) oder an eine bekannte Sprache angelehnt (z.B. Methodendeklarationen in Java). Die Semantik wird durch einen aufgabenspezifischen Interpreter realisiert. Dieser verarbeitet die studentische Einsendung und liefert (sofort) diese Informationen:

- Einsendung ist richtig oder falsch bzgl. Aufgabenstellung
- wenn richtig, eine Meßgröße (z.B. Eingabegröße) für eine Highscore-Wertung
- wenn falsch, eine Spur der ausgeführten Rechnung (Probe) mit Teilschritten.

Der Einsender soll aus der Fehlermeldung (Spur der Probe) und dem Wissen aus der Vorlesung erschließen, wie die Einsendung zu reparieren ist, und kann das innerhalb der vorgesehenen Bearbeitungsfrist beliebig oft versuchen. *autotool* unterscheidet sich damit grundlegend von E-„Learning“-Systemen, bei denen doch nur überprüft wird, ob an vorgegebenen Stellen Kreuzchen gemacht wurden.

Die Erfahrung zeigt, daß die Studenten vor allem die sofortigen, ausführlichen und nachvollziehbaren Rückmeldungen des Systems schätzen und die sich daraus ergebende Möglichkeit, zu beliebiger Tages- (und Nacht-)Zeit Aufgaben zu bearbeiten.

Zu vielen Aufgabentypen gibt es Instanzen-Generatoren. Diese erzeugen nach einstellbaren Parametern verschiedene, aber ähnlich schwere Aufgabeninstanzen eines Typs. Jeder Student erhält eine eigene Aufgabeninstanz und kann deren Lösung deswegen nicht bei anderen abschreiben.

Für Aufgaben mit gemeinsamer Instanz kann eine öffentliche (und pseudonymisierte) Highscore-Liste geführt werden, in der allen richtigen Einsendungen nach der Meßgröße sortiert werden und jene mit übereinstimmenden Werten nach Einsendezeitpunkt. Wer einen vorderen Platz in dieser Liste ergattern will, wird auch niemanden abschreiben lassen.

Im vorliegenden Bericht beschreibe ich einige der von mir entwickelten, implementierten und in Vorlesungen angewendeten Aufgabentypen zu Prinzipien von Programmiersprachen. Dabei gebe ich jeweils an, welchem Zweck die Aufgabe dienen soll, dann eine exakte Spezifikation von Aufgabenstellung und Korrektheit der Lösung, sodann ein Beispiel für eine Aufgabeninstanz mit Lösung. Es folgen jeweils Beispiele für Systemantworten bei inkorrekten Einsendungen, denn das ist der wohl der häufigste Anwendungsfall und somit ein wesentlicher Bestandteil des Lernprozesses. Beispiel-Aufgaben können ausprobiert werden unter `https://autotool.imn.htwk-leipzig.de/cgi-bin/Trial.cgi?lecture=213`.

Der hier vorgelegte Artikel erscheint ähnlich in den informellen Pre-Proceedings des *Workshops E-Learning* Leipzig 2015.

Ich bedanke mich bei Hans-Gert Gräbe für nützliche Diskussionen zu Vorstufen dieses Aufsatzes sowie bei vielen Teilnehmern meiner Vorlesungen in den letzten Jahren für den (unfreiwilligen) Test der Aufgaben.

2 Aufgabe: Polymorphe Typisierung

Die parametrische (generische) Polymorphie ist ein wichtiges Hilfsmittel zur Software-Wiederverwendung. Polymorphe Typen werden oft für Collections verwendet, der Typparameter ist dann der Elementtyp.

2.1 Spezifikation

- Instanz: eine polymorphe Signatur und ein Typ T
- Einsendung: ein Term t
- Bedingung: t ist korrekt typisiert und hat Typ T.
- Highscore-Parameter: die Größe von t

Die Syntax ist sehr stark an Java angelehnt. Bei jedem Aufruf einer Methode mit polymorphem Typ müssen jedoch alle Typargumente explizit angegeben werden — auch wenn das in Java (meist) inferiert werden könnte.

Bei gegebener monomorpher Signatur bilden die korrekt typisierten Terme eine reguläre Baumsprache. In polymorphen Signaturen ist die Lage viel komplizierter, man kann etwa die Lösung eines Postschen Korrespondenzproblems in der Typ-Ableitung kodieren. Das ergibt anspruchsvolle Highscore-Aufgaben. . . .

2.2 Beispiel mit Lösung

```
Gesucht ist ein Ausdruck vom Typ Fozzie<Kermit, Kermit>
in der Signatur
    class S {
        static <T2> Piggy<Piggy<Animal>> statler ( Piggy<T2> x
                                                 , Piggy<T2> y );
        static <T2> Kermit waldorf ( Piggy<T2> x );
        static Piggy<Fozzie<Animal, Animal>> bunsen ( );
        static <T2, T1> T1 chef ( Piggy<Piggy<T2>> x
                                , Piggy<Piggy<T1>> y );
        static <T2> Fozzie<Kermit, T2> rowlf ( T2 x , Animal y );
    }
```

Der Zieltyp ist der Resultattyp von `S.<Kermit>rowlf`. Um das aufzurufen, braucht man einen `Kermit` (erhält man von `waldorf`, dessen Argumenttyp ist egal, also kann man `bunsen()` nehmen) und ein `Animal`:

```
S.<Kermit>rowlf(S.<Fozzie<Animal, Animal>>waldorf( S.bunsen())   ,
  S.<Animal,Animal>chef(
   S.<Fozzie<Animal,Animal>>statler( S.bunsen(),S.bunsen() ),
  S.<Fozzie<Animal,Animal>>statler( S.bunsen(),S.bunsen() ) ) )
```

2.3 Typische Fehlermeldungen

```
berechne Typ für Ausdruck: S.<Fozzie<Animal, Kermit>>statler (S.bunsen (),
                          S.bunsen ())
    Name statler hat Deklaration:
      static <T2> Piggy<Piggy<Animal>> statler ( Piggy<T2> x, Piggy<T2> y )
    die Substitution für die Typ-Parameter ist
        listToFM
            [ ( T2 , Fozzie<Animal, Kermit> ) ]
    die instantiierte Deklaration der Funktion ist
        static Piggy<Piggy<Animal>> statler ( Piggy<Fozzie<Animal, Kermit>> x
                                            , Piggy<Fozzie<Animal, Kermit>> y
                                            )
    prüfe Argument Nr. 1
        berechne Typ für Ausdruck: S.bunsen ()
```

```
            Name bunsen hat Deklaration:
                static Piggy<Fozzie<Animal, Animal>> bunsen ( )
        Ausdruck: S.bunsen ()
        hat Typ: Piggy<Fozzie<Animal, Animal>>
    Argument-Typ stimmt mit instantiierter Deklaration überein?
        Nein.
```

2.4 Instanzen-Generator

Hier die Konfiguration des Generators, mit der obige Instanz erzeugt wurde:

```
Conf
    { types_with_arities = [ (Kermit,0), (Animal,0), (Piggy,1) , (Fozzie,2) ]
    , type_variables = [ T1 , T2  ]
    , function_names = [ statler , waldorf , bunsen, chef, rowlf ]
    , type_expression_size_range = ( 1 , 4 ) , arity_range = ( 0 , 2 )
    , solution_size_range = ( 6 , 12 ) , generator_iterations = 500
    , generator_retries = 10
    }
```

Bei der Generierung einer Instanz wird zunächst eine Signatur gewürfelt und dann korrekt typisierte Terme in dieser Signatur der Größe nach aufgezählt. Dieser Vorgang wird einigemale wiederholt und schließlich die Instanz gewählt, die den gewünschten Parametern am nächsten kommt.

3 Aufgabe: Semantik rekursiver Programme

Die operationale Semantik beschreibt die Rechenschritte, mit denen ein Programm ausgewertet wird. Die denotationale Semantik ordnet den Bezeichnern eines Programmtextes mathematische Objekte zu. Den Unterschied kann man für (gegenseitig) rekursive Definitionen von Funktionen so erklären: operational ist ein Programm ein Ersetzungssystem, denotational beschreibt es Funktionen. Wenn man Glück hat, sind diese eindeutig und total.

In Sonderfällen besitzen rekursive Gleichungssysteme eine explizite Darstellung der Lösung durch einfache arithmetische Funktionen und Verzweigungen. Zum Beispiel zeigt man für $g(n) = \text{if } n > 0 \text{ then } 2 + g(n-1) \text{ else } 1$ leicht $g(n) = 2n+1$. In ausgesuchten anderen Fällen [Knu91] ist das ebenfalls möglich, aber überhaupt nicht offensichtlich.

3.1 Spezifikation

- Instanz: ein Gleichungssystem E, möglicherweise rekursiv, für unbekannte Funktionen $f_1, \dots, f_n$,
- Einsendung: explizite arithmetische Ausdrücke $A_1, \dots, A_n$
- Bedingung: die Substitution $\sigma : f_1 \mapsto A_1, \dots$ ist eine Lösung von E
- Highscore-Parameter: $\sum |A_i|$

Die Bedingung ist nicht entscheidbar, deswegen wird wiederholt getestet. Beim Test wird *nicht* die Einsendung mit einer bekannten Lösung verglichen, *sondern* die eingesandte Substitution σ wird auf alle Gleichungen von E angewendet und die dabei entstehenden Gleichungen zwischen arithmetischen Ausdrücken werden getestet — und das geht schnell. Selbst wenn E rekursiv ist, findet beim Testen einer Einsendung keinerlei Rekursion statt.

Es ist Sache des Aufgabenstellers, darauf zu achten, daß das Gleichungssystem eine geschlossen darstellbare Lösung besitzt. Eindeutige Lösbarkeit ist nicht erforderlich.

3.2 Beispiel mit Lösung

Die Takeuchi-Funktion kann so beschrieben werden:

```
Deklarieren Sie Funktionen mit den folgenden Eigenschaften.
wobei die Variablen über alle natürlichen Zahlen laufen:
    {forall x y z . t (x  , y  , z ) ==
    if x  <= y  then y
    else t (t (x  - 1 , y  , z ) ,
           t (y  - 1 , z  , x ) ,
           t (z  - 1 , x  , y ))
    ;}
```

Eine explizite Lösung lautet

```
{ t(x,y,z) =
  if x <= y then y else if y <= z then  z else x ;}
```

3.3 Typische Fehlermeldungen

```
gelesen: {t (x , y , z) = if x  <= y then y else z  ;}
nicht erfüllte Bedingungen:
    Constraint: forall x y z .
                t (x  , y  , z ) == if x  <= y
                then y
                else t (t (x  - 1 , y  , z ) ,
                       t (y  - 1 , z  , x ) ,
                       t (z  - 1 , x  , y ))
                ;
    Belegung: x = 3 ; y = 2 ; z = 1 ;
        dabei berechnete Funktionswerte:
        t (3 , 2 , 1) = 1
        t (2 , 2 , 1) = 2
        t (1 , 1 , 3) = 1
        t (0 , 3 , 2) = 3
        t (2 , 1 , 3) = 3
```

4 Aufgabe: Approximierte Spur-Semantik

Gegenstand ist die Semantik von imperativen Programmen als Menge von möglichen Ausführungen (Spuren) und deren Anwendung beim Feststellen der Äquivalenz verschiedener Arten der Programmablaufsteuerung (wie Schleifen und Sprünge).

4.1 Spezifikation

- Instanz: ein imperatives Programm P, eine Beschreibung von erlaubten Steuerbefehlen, z.B. „nur While (kein Goto)“, „nur Goto (kein While)“.
- Einsendung: ein imperatives Programm Q,
- Bedingung: Q hat erlaubte Struktur und ist spur-äquivalent zu P
- Highscore-Parameter: Größe von Q

Programme bestehen dabei aus abstrakten elementaren Befehlen. Diese sind benannt, aber ihre Semantik ist nicht spezifiziert. Die Programmablaufsteuerung benutzt boolesche Kombinationen abstrakter Zustandsprädikate. Diese sind ebenfalls nur benannt, aber nicht spezifiziert.

Eine Spur ist eine Folge von Paaren von Zustand und Befehl. Ein Zustand ist eine Zuordnung von Prädikat-Namen zu Wahrheitswerten. Die Spuren des Programmes `if P then A; B;` sind `[(P,A), (P,B)]`, `[(P,A), (not P, B)]` und `[(not P,B)]`. Dadurch wird modelliert, daß jede Bedingsauswertung nebenwirkungsfrei ist und jede Befehlsausführung jede Zustandsinformation zerstört. Die Menge der Spuren eines Programmes ist in diesem Modell effektiv regulär und die Spuräquivalenz damit entscheidbar. Sind die so definierten Spursprachen von P und Q gleich, so sind alle Instantiierungen von P und Q als konkrete imperative Programme äquivalent.

Unter diesem strengen Äquivalenzbegriff kann nicht jedes Goto-Programm in ein äquivalentes While-Programm transformiert werden, da man keine (booleschen) Variablen zur Verfügung hat, um Wissen über frühere Zustände abzulegen.

4.2 Beispiel mit Lösung

```
Gesucht ist ein Programm,
das äquivalent ist zu
    {foo : while (a)
               {while (b)
                   {p;
                   if (c) continue foo;
                   q;}}}
und diese Bedingungen erfüllt
    And    [ Flat , No_Loops ]
```

Lösung:

```
{foo : if (!a) goto foo_end;
 bar: if (!b) goto bar_end;
 p;
 if (c) goto foo;
 q;
 goto bar;
 bar_end: skip;
 goto foo;
 foo_end: skip;}
```

4.3 Typische Fehlermeldungen

```
gelesen: {foo : if (! a) goto foo_end;
         bar : if (! b) goto bar_end;
         p;
         if (c) goto bar;
         q;
         goto bar;
         bar_end : skip;
         goto foo;
         foo_end : skip;}

Ist Spursprache des Programms aus Aufgabenstellung
Teilmenge von Spursprache des Programms aus Ihrer Einsendung ?

Nein. Wenigstens diese Wörter sind in
   Spursprache des Programms aus Aufgabenstellung
, aber nicht in
   Spursprache des Programms aus Ihrer Einsendung :
    [ [ ( state [(a,True),(b,True),(c,True)] , Execute    p )
      , ( state [(a,False),(b,True),(c,True)] , Halt ) ] ]
```

5 Aufgabe: Statische und dynamische Ketten

Zur Verwaltung von Unterprogramm-Aufrufen benutzt man Frames, die durch Zeiger verbunden sind. Der dynamische Vorgänger eines Frames ist der Frame des aufrufenden Unterprogramms und wird zur Programmablaufsteuerung benötigt. Der statische Vorgänger ist der Frame des textuell umgebenen Unterprogramms (genauer: der Frame, in dem die Closure konstruiert wurde) und dient dem Zugriff auf die Werte lokal gebundener Namen.

Lokalen Unterprogramme (Lambda-Ausdrücke) in Sprachen wie C#, Javascript und Java unterstreichen die Bedeutung dieses Konzeptes auch in der sogenannten Mainstream-Programmierung.

Die Übungsaufgabe besteht darin, einen Programmtext so zu vervollständigen, daß bei Ausführung eine vorgegebene Struktur aus dynamischen und statischen Zeigern entsteht.

Die Programme werden in (stark eingeschränktem) Java-Script notiert. Es gibt lokale Unterprogramme, formale Parameter und lokale Variablen, aber keine Bedingungsauswertung.

5.1 Spezifikation

- Instanz: ein gerichteter Kanten-2-gefärbter Graph $G = (\{1, \dots, n\}, E_1, E_2)$, ein Lücken-Programmtext S
- Einsendung: ein vollständiges Programm P
- Bedingung: P paßt zu S und bei Ausführung von P entstehen der Reihe nach die Frames $1, \dots, n$ mit statischer Vorgänger-Relation E_1 und dynamischer Vorgänger-Relation E_2.
- Highscore-Parameter: Größe von P

Tatsächlich sind nicht viele Graphen G so realisierbar. Jede Kante zeigt in die Vergangenheit und ist unveränderlich. Sowohl der statische als auch der dynamische Teilgraph sind also Bäume. Deren Gestalten können sich jedoch erheblich unterscheiden.

5.2 Beispiel (ohne Lösung)

```
Ersetzen Sie im Programm
    { missing ; missing ;  missing ; }
jedes 'missing' durch eine Deklaration oder einen Ausdruck,
so daß nach höchstens 25 Auswertungsschritten
die Anweisung 'halt' erreicht wird
und die Frames dann folgende Verweise enthalten:
    Frame 1 : dynamischer Vorgänger 0 , statischer Vorgänger 0 ;
    Frame 2 : dynamischer Vorgänger 1 , statischer Vorgänger 1 ;
    Frame 3 : dynamischer Vorgänger 1 , statischer Vorgänger 1 ;
    Frame 4 : dynamischer Vorgänger 3 , statischer Vorgänger 2 ;
    Frame 5 : dynamischer Vorgänger 1 , statischer Vorgänger 4 ;
```

Diese Aufgabe ist lösbar — probieren Sie es!

5.3 Typische Fehlermeldungen

Häufige semantische Fehler in Einsendungen sind: es wird ein nicht deklariertes (nicht sichtbares) Unterprogramm aufgerufen; der Frame-Graph stimmt nicht mit G überein. — Beispiel (Ausschnitte):

```
gelesen: { f = function ( ) { halt ; };
         g = function ( y ) {
            h = function ( i ) { i ( ); }; h ( f );
          };
          g ( 42 ); }
```

```
Schritt 1 (in Frame 1)
 beginne Auswertung von function ( ) { halt ; }
    Ergebnis von Schritt 1 ist
      ValClosure { link = 1
                 , body = function ( ) { halt ; } }
Schritt 2 (in Frame 1)
  beginne Auswertung von
    function ( y ) { h = function ( i ) { i ( ); }; h ( f ); }
    Ergebnis von Schritt 2 ist
      ValClosure { link = 1
        , body = function ( y )
          { h = function ( i ) { i ( ); }; h ( f ); }
        }
...
Dieser Speicherzustand wird erreicht:    Store
  { step = 11
  , max_steps = 25
  , store = listToFM
     [ ( 1 , Frame { number = 1, dynamic_link = 0, static_link = 0 , ... } )
     , ( 2 , Frame { number = 2, dynamic_link = 1, static_link = 1, ... } )
     ...
     ]
  }
...
der dynamische Vorgänger soll 1 sein:
  Frame { number = 3 , dynamic_link = 2 , static_link = 2 }
```

6 Bemerkungen zur Implementierung

Abschließend betone ich einige technische Aspekte bei der Realisierung der vorgestellten Aufgaben sowie deren Auswirkungen auf die Didaktik.

Die Implementierungssprache des autotool ist Haskell [Mar10]. Die hohe Ausdrucksstärke und Typsicherheit der Sprache erleichtert (eigentlich: ermöglicht) das effiziente Schreiben von Parsern und Interpretern für domainspezifische (d.h. hier: aufgabenspezifische) Sprachen — und hat weitere Vorteile.

Studentische Einsendungen erfolgen ausschließlich in textueller Form, es gibt absichtlich keinerlei Unterstützung für grafische Editoren, jedoch folgende textuelle Eingabehilfen: Das Texteingabefeld ist typisiert. Aus dem Typ der Eingabe wird der zu benutzende Parser bestimmt (realisiert mit `parsec` [LM01], bei Syntaxfehlern werden automatisch die möglichen nächsten Token angezeigt) sowie (durch compile-time reflection mit `Data.Typeable`) ein URL erzeugt und angezeigt, der auf die API-Dokumentation des Eingabetyps verweist. Diese wurde mit `haddock` [MW10] erzeugt und enthält Verweise auf die tatsächlichen Quelltexte.

Studenten werden dadurch angehalten, Typdeklarationen und Quelltext zu lesen und sollen dabei erkennen, daß beides sehr weitgehend den Definitionen an der Tafel entspricht. Sie werden damit auch auf die (optionale) Compilerbau-Vorlesung [Wal13] vorbereitet, wo sie domainspezifische Interpreter selbst schreiben.

Literatur

Knu91. Donald E. Knuth. Textbook Examples of Recursion. `http://arxiv.org/abs/cs/9301113`, 1991.

LM01. Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

Mar10. Simon Marlow. Haskell 2010 Language Report. `https://www.haskell.org/onlinereport/haskell2010/`, 2010.

MW10. Simon Marlow and David Waern. Haddock User Guide. `https://www.haskell.org/haddock/`, 2010.

RW02. Mirko Rahn and Johannes Waldmann. The Leipzig autotool System for Grading Student Homework. In Michael Hanus, Shriram Krishnamurthi, and Simon Thompson, editors, *Functional and Declarative Programming in Education (FDPE 2002)*, 2002. `http://www.informatik.uni-kiel.de/~mh/publications/reports/fdpe02/`.

Wal13. Johannes Waldmann. M****** in der Compilerbauvorlesung. In *30. Workshop der GI-Fachgruppe Programmiersprachen und Rechenkonzepte, Bad Honnef*, 2013. `http://www.imn.htwk-leipzig.de/~waldmann/talk/13/fg214/`.

Access Control for Weakly Consistent Data Stores

Mathias Weber and Annette Bieniusa

University of Kaiserslautern, Germany
{m_weber, bieniusa}@cs.uni-kl.de

Abstract. Information systems have become distributed over large distance networks to serve an ever-increasing demand for fast data access at global scale. This trend lead to a growing popularity of NoSQL data stores as they provide better performance and response times in the distributed setting than standard relational databases. To achieve these properties, these data stores sacrifice consistency guarantees for the data stored in favor of availability and robustness under network failures and partitions. Consequently, it is more difficult to secure such systems against unauthorized data access without introducing performance bottlenecks. Due to the weak consistency guarantees, it became much harder to build access control systems coupled with the data store because the policies are replicated and can become inconsistent. Using a separate strongly consistent system to implement access control seems more feasible, but this architectural design adds additional complexity and results in performance loss and single points-of-failure.
In this paper, we outline the challenges when building access control systems for distributed information systems based on weakly consistent data stores. Based on a formal model, we present a solution that correctly applies access control policies. It guarantees convergence of policy modifications that are concurrently issued at different datastore replicas.

Keywords: access control, security, weak consistency

1 Introduction

Traditionally, information systems where built in a centralized fashion with a single replica of all data. Information systems today are distributed all over the globe to provide fast response times and low latency. However, the techniques for developing information systems has drastically changed in the last years. This change was triggered by new trends such as the Internet of Things and Industry 4.0 as well as the rapid growth of the Internet.

These demands have lead to an increasing popularity of weakly consistent data stores. The main focus of these data stores is availability and fast response times they achieve by sacrificing the strong consistency guaranties that are at the core of traditional relational database systems. One consequence of this sacrifice is that the semantics of data stores have become more complex because weaker consistency guaranties allow more interleaving of operations, which leads to surprising behavior developer might be unaware of. Recently[4, 5, 12, 13], Replicated Data Types have gained a lot of interest, partly also in industry. But the question of how to implement access control on weakly consistent data stores is an open question.

Access control systems guarantee that every action performed adheres to a set of rules, which can be dynamically changed at runtime. In traditional systems, this guarantee can be enforced by relying on a central server. This server fixes a total order of the operations which avoids conflicts. Using such a centralized architecture is not possible in a highly available, globally distributed system that requires low latency. A central access control server introduces a severe bottleneck and increases the total latency of all actions in the system. To reduce the latency we can sacrifice part of the consistency guarantees offered by the access control system. Gilbert and Lynch [8] have shown that high availability, partition tolerance and strong consistency cannot be achieved by any distributed system at the same time. This theorem is also known as the CAP theorem. One possible solution is to not handle the policies by a central server, instead they are replicated to different servers. This introduces a security threat since the rules can be modified on different servers in non-consistent ways. The access restrictions are based on the local copy of these policies, which can be outdated.

We show the importance of the causal relation between data operations and access control operations for the correctness of an access control system (Section 3). We describe the design-space of access control systems for weakly consistent data stores with replication (Section 4). We created a formal model for such a system which works applicative and achieves convergence of the policies of different replicas and sketch the most important proofs of the model including the proof of eventual consistency (Section 5). The model is formalized in Isabelle/HOL.

2 Information System Model

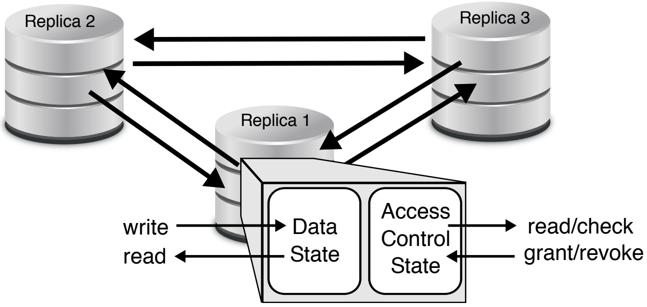


Fig. 1. Overview of the system model

We model an information system as a set of replicas where each replica represents a server with a local copy of the data managed by the information system.

Figure 1 shows an overview of the system. The replicated data consists of the data state and the access control state. The data state consists of the cooperative operations which make up the state of the part of the world represented in the information system. The state can be accessed using *read* and *write* operations. The access control state consists of access control policies, which influence the decision of the access control system. Policies in the access control state can be changed using the *grant* and *revoke* operations, which grant and revoke rights of a user to read and modify the data state and to grant or revoke rights of other users. In addition, the current set of policies can be inspected and system operations can be checked for compliance with the current policies using the *check* operation. Each replica has a local access control system allowing or disallowing execution of local operations on the replica. The decision of the access control system is only based on the local copy of the access control state. We assume the level of rights to be the level of replicas which means in order to restrict the rights of individual users each user has to work on his/her own replica. A generalization of this model to individual users and more complex access control policy patterns such as groups is left as future work.

Operations performed by a replica are broad-cast to the other replicas using messages. Each messages can be uniquely identified and caries only a single data or access control operation. Sending and receiving messages is asynchronous, so the sending replica does not wait for the message to be accepted by all replicas. To simplify things we assume reliable transport of messages, which means message loss has to be compensated on the network level, and full replication, that is every replica has a full copy of the data available.

3 Causal Consistency

To illustrate the importance of the causality of operations we start with an example system. Consider a social network where users can create a personal page, galleries for uploaded pictures and it is possible to invite friends to look at your personal data. By default, the system denies access to your personal page to all users. Other users can be added to a friend list. Users on this friend list have full access to all personal data including the posts on the personal page as well as all galleries.

Anne has an existing list of friends. One of these friends is Paul. After a heated discussion with Paul, Anne removes him from her friend list before uploading her new photos of the last party. Anne does not want Paul to have access to her new photos after removing him as a friend.

There is a causal relation between the remove operation of Paul and the upload of the new photos. In a system with strong consistency guarantees, it would not be possible for Paul to access the new photos as long as he is not readded to the friend list of Anne. When considering a replicated system which does not retain the causal relation between operations, there might be a server that receives the upload of the photos before the update of the friend list. This

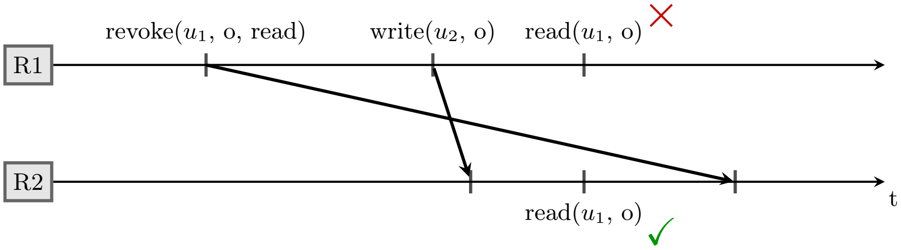


Fig. 2. Undesired semantics because of causality violation.

gives Paul the possibility to access the new photos before the change to the friend list of Anne is known to this local server.

As we see in the example above, it is important for the correctness of an access control system to preserve the causal relation between operations. We can distinguish several cases: (1) an operation on some data is performed because it is allowed by the current policies of the system; (2) the application computes new values based on the data state it sees in the system; and (3) the application changes the policies of the access control system based on previous policy changes.

Case (1) sketches the usual operation of the system: The data operations in the system are checked by the access control system to comply with the current policies. Only operations allowed by the access control policies may be performed by the system.

Case (2) describes the normal operation of the system without access control. Every computation reads values from the system, computes new values and enters them into the system. This relation between the values read and the values entered should be kept intact by the data store.

The last case (3) is how access control policies evolve. The initial state of the access control system consists of an initial set of policies. These policies can be adapted over time, for example because responsibilities and roles of persons change.

The causality between data operations can be loosened depending on the data types used and the guarantees needed by a specific application. Loosening the causality between access control policy changes and data operations invalidates the guarantees an access control system should offer.

4 Distributed Access Control

When designing an access control system for a distributed system there are some issues that cannot be avoided. We sketch these issues and their influence on the access control system.

As discussed in Section 3, the causal relation between access control policy changes and data operations have to be retained. The problem is illustrated in

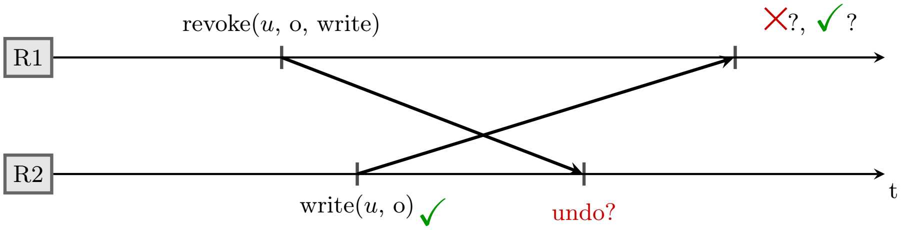


Fig. 3. Possible inconsistencies because of local decision.

Figure 2. There are existing data stores that offer causal consistency guaranties such as SwiftCloud [9] and Antidote [1]. Therefore, we can assume in our model of the access control system that the data store offers causal consistency as an option and will not go into the details of implementing causal consistency for the access control system. Even though the causal relation between access control operations and data operations is important, it is not sufficient for the convergence, and thus for the correctness of the access control system.

Since the access control state is local to the replica and synchronized using message broadcast, the access control state of different replica is temporarily inconsistent. Figure 3 shows such a scenario. Replica R1 performs an operation which revokes the right of user u to perform *write* operations on object o. This change of the access control policies is transmitted to replica R2 using a message. While this message was not yet accepted by R2, u tries to write to o, which is accepted by R2 according to the current local version of the policies. If the same operation would have been performed on R1, the access control system on R1 would have denied the operation. When the revoke message arrives at R2, it becomes clear that the access control state was inconsistent.

To avoid such inconsistencies, there are two possible solutions. Receiving the revoke operation on R2 could result in undoing of the *write* operation. This approach is favored by Cherif et al. [6] and Samarati et al. [10] and is known as optimistic approach. Alternatively, the *write* operation takes priority over the *revoke* operation, since the *revoke* was not yet accepted while executing the *write* on R2 and cannot easily be undone. This interprets the *write* operation as if it would have happened before the *revoke* operation. This approach needs access to the history of previous access control policies, since R1 has to check the validity of the *write* operation before accepting it[1].

One design decision to be made for a replicated access control system is whether to trust the other replicas or to put the trust only in the access control policies. When not trusting in the correctness and reliability of replicas, the broadcast messages have to be checked and accepted by the local access control system in order to protect the system from malicious actions. On the other hand, expecting the replicas to be distrusted makes it possible to place replicas in an

[1] We expect the data state to converge with the help of a different strategy such as convergent replicated data types

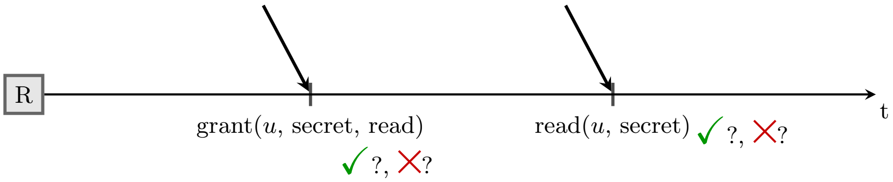


Fig. 4. Missing checks of remote messages.

untrusted environment such as the computer of a user. Figure 4 shows a simple attack scenario. An attacker u can run a manipulated replica which sends a forged message to replica R. Using this message, u can try to grant himself the right to read a secret object, even though u might not have the right to perform this policy change. If the message was accepted by R, this would lead to a set of access control policies which would grant a future *read* access to the *secret* object by u until the policies are repaired. This inconsistency is not acceptable with respect to the semantics of the access control system. Wobber et al. [14] describe a system that uses public and private key certificates to achieve trust in policies changes. A different way is to only accept policy changes that can be deduced to be allowed by the default policy set and already accepted changes.

The access control state has to converge on all replicas. Since the policies are changed locally and forwarded asynchronously, we cannot directly avoid inconsistent changes to the policies by different replicas. Other systems [6, 10] solve this problem by assigning one replica as the owner of an object. Only the owner replica can make the final decision about policy changes regarding the object owned. Since we want to support partitioning for the network, we have to follow a different direction. If, for example, we consider groups in social networks, the members of these groups may be distributed over the world and specific replicas may be separated because of network errors. Fixing a replica which may change the membership of a group would lead to unacceptable down-times.

If we accept that multiple replicas may change the access control policies for each object we also have to accept that there will be a window where the policies will not be consistent. The messages that distribute these changes to the access control state of remote replicas can arrive in any order because of the concurrent nature of the system and the delay caused by the network. Therefore we need a merge algorithm that incorporates remote changes into the local policies such that the outcome on all replicas is the same after processing all pending messages. We call this property the *convergence* of the access control state.

The naive approach of just accepting policy changes as they arrive at a replica can lead to different decisions on two replicas. Figure 5 shows the problem. The messages have to be extended by meta-data which allows to order the messages and this order is the same on all replicas.

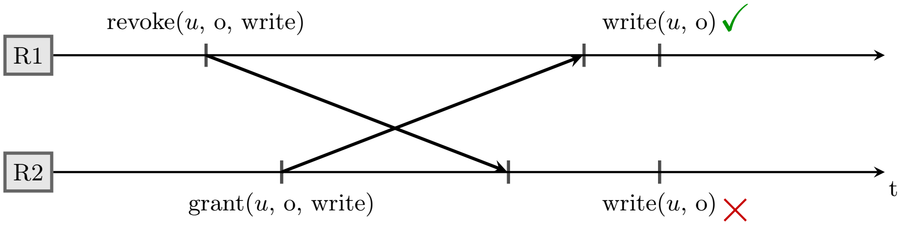


Fig. 5. Merging Policy Changes

5 Access Control Model for Distributed Information Systems

In the previous section we described the design space of implementing an access control system for globally distributed replicated information systems. In the following, we present a solution.

5.1 Formal Model of the Data Store

Replicas	$r \in R$
Data operations	$op_D \in Op_D = R \times OT \times \mathbb{N}$
Admin. operations	$op_{AC} \in Op_{AC} = AOT \times R \times OT \times \mathbb{N}$
Messsages	$msg \in M = R \times Op_D \cup Op_{AC}$
Global State	$gs \in R \times RS \times opDeps$
Replica state	$rs \in RS = \overline{Op_D} \times \mathcal{P}\left(Op_{AC}\right) \times \overline{M} \times \mathcal{P}\left(M\right)$
Dependencies	$opDeps :: M \mapsto \mathcal{P}\left(M\right)$
Operation Type	$ot \in OT$
Adm. Context	$actx \in \mathbb{N}$
Adm. Op.	$aot \in AOT = \{Grant, Revoke\}$

Fig. 6. Definitions used in the formal model

The over-bars symbolize sequences. For example $ms \in \overline{M}$ stands for a sequence of messages where the ith element can be accessed by $ms@i$.

We assume the set of replicas to be fixed, meaning neither can new replicas be added to the system nor can existing replicas be removed. Further, we simplify the model by assuming that each user works on his/her own replica.

State The global *data store* state gs consists of a set of replica (replicas(gs)), their local states (replicaState(gs)) as well as the operation dependencies (called operationDeps(gs)). The *operation dependencies* is a mapping from a message m to the set of messages *deps* that m causally depends on. The *local state* of a replica consists of the data state (persistentOps(ls)), in form of a list of data operations that have been performed on the replica, and the access control state ($admOps(ls)$), in form of a set of administrative operations or policy changes performed on the replica. In addition to that, the local state also consists of a list of accepted message (acceptedMessages(ls)) and an incoming queue (incomingQueue(ls)), the queue of messages that have been sent by other replicas but were not yet accepted by the replica. The incoming queue is modeled as a set to reflect the non-determinism involved in transferring messages over a switching network. In this way the order in which the broadcast messages are handled by each replica is not fixed by the incoming queue.

Operations A *data operation* $op_D = (r, ot, actx)$ consists of the origin replica r the operation was first performed on, the operation type ot, which can also include the target object of the operation, and the administrative context $actx$. The administrative context is needed by the access control system to determine the policy that allowed this operation on the origin replica. An *administrative operation* $op_{AC} = (aot, r, ot, actx)$ consists of the type of the administrative operation aot, either *Grant* or *Revoke*, the replica r and operation type ot this policy change regulates and the administrative context. The messages distributing the changes to other replicas are *data messages* $msg_D = (r, op_D)$ and *administrative messages* $msg_{AC} = (r, op_{AC})$ and each consist of the sending replica and the operation to be transfered. We assume that each message can be uniquely identified and from the models point of view no message is sent or received twice.

Initial State When starting-up for the first time, the system is in its *initial state*, which means all replicas are in initial local state, there are no dependencies between messages and the policies are equal for all replicas. These initial policies can for example state that the root-user (in our case the root-replica) may assign rights to other users (replicas) and all other users have no rights at all. A replica is in *initial local state* if the data state and the access control state is empty and no messages have been accepted and the incoming queue is empty.

Access Control Policies Before we come to the steps such a system can make we first have to clarify what the access control policies are. The *default policies* apply when no other policy in admOps(ls) applies for the operation on the replica. The access control policies of a replica state is the set of default policies plus the set of administrative operations processed by a replica

$$\mathrm{acPolicies}(ls) = \mathrm{defaultPolicies} \cup \mathrm{admOps}(ls)$$

Each administrative operation carries an administrative context which is an element of a totally ordered kind such as the natural numbers. This context is like a version number for policy updates. The *relevant policy* for an operation on a replica $\text{relevantPolicy}(r, ot, ls)$ is a policy $p = (aot_p, r_p, ot_p, actx_p)$ where $r_p = r$ and $ot_p = ot$ and

$$\begin{aligned}
&p \in \text{acPolicies}(ls) \wedge \\
&\forall\ (aot'_p, r'_p, ot'_p, actx'_p) \in \text{acPolicies}(ls).\, (r'_p = r \wedge ot'_p = ot) \implies actx_p \geq actx'_p
\end{aligned}$$

In other words: The relevant policy is the policy in the set of the access control policies with the greatest access control context. A data message is *allowed to be sent* by a replica $\text{sendOK}(ls, msg_D)$ where $msg_D = (r, op_D)$ and $op_D = (r, ot, actx)$ if the relevant policy for this message allows the operation in the current administrative context

$$\text{relevantPolicy}(r, ot, ls) = (Grant, r, ot, actx)$$

The operation is performed on r and broadcast to the other replicas.

A administrative message is *allowed to be sent* by a replica $\text{sendOK}(ls, msg_{AC}$ where $msg_{AC} = (r, op_{AC})$ and $op_{AC} = (aot, r, ot, actx)$ if the relevant policy is the inverse of the operation

$$\text{relevantPolicy}(r, ot, ls) = (\neg aot, r, ot, actx - 1)$$

The inverse of *Grant* is *Revoke*, $\neg Grant = Revoke$ and the other way round $\neg Revoke = Grant$.

The *allowing policy* for a data message $\text{allowingPolicy}(msg_D)$ where $msg_D = (r, op_D)$ and $op_D = (r, ot, actx)$ is the corresponding policy $(Grant, r, ot, actx)$. r in this case is the replica which originally executed the operation first and also the sender of the corresponding message. The *allowing policy* for an administrative messages $\text{allowingPolicy}(msg_{AC})$ where $msg_{AC} = (r, op_{AC})$ and $op_{AC} = (aot, R, ot, actx)$ is $(\neg aot, r, ot, actx - 1)$. This means an administrative operation may only be performed if the inverse policy with the previous administrative context is already part of the current policy set. This construction enforces that the administrative context is monotonically increasing for each policy change and therefore makes each relevant policy unique. A message is *allowed to be processed* $\text{processOK}(ls, msg)$ if the allowing policy is in the local access control policy set $\text{allowingPolicy}(msg) \in \text{acPolicies}(ls)$.

We expect the data store to be causally consistent. A data store is *causally consistent* if for all messages accepted by a replica all dependencies have been accepted before. We write $l@i$ to denote the ith element of list l.

$$\begin{aligned}
&\text{causallyConsistent}(gs) \equiv \\
&\forall r \in \text{replicas}(gs), \forall rs = \text{replicaState}(gs, r), \forall m1, i. m1 = \text{acceptedMessages}(rs)@i, \\
&\quad \forall m2 \in \text{operationDeps}(gs, m1).\, \exists i' < i.\, \text{acceptedMessages}(rs)@i' = m2
\end{aligned}$$

A message is *causally ready* on a replica if all its dependencies have already been accepted.

$$\begin{aligned}&\text{causallyReady}(gs, msg, ls) \equiv \forall m \in \text{operationDeps}(gs, msg)\\&\quad \exists i.\, \text{acceptedMessages}(rs)@i = m\end{aligned}$$

Steps With these definitions we can define the possible steps how the system can evolve: When doing a *step from one global state to the next* $gs \longrightarrow gs'$, we can either accept a message from the incoming queue $gs \xrightarrow{accept} gs'$, or a replica can perform an operation and send a new broadcast message $gs \xrightarrow{send} gs'$. A message msg may only be accepted by a replica r with replica state rs if

$$\begin{gathered}msg \in \text{incomingQueue}(rs) \wedge \text{causallyReady}(gs, msg, rs) \wedge\\ \text{acceptMessage}(msg, rs, rs') \wedge \text{processMessage}(msg, rs, rs')\end{gathered}$$

The message may only be *accepted* if it is causally ready and currently in the incoming queue. It is accepted by removing it from the incoming queue and appending it to the list of accepted messages of the replica.

$$\begin{aligned}&\text{acceptMessage}(msg, rs, rs') \equiv\\&\quad \text{acceptedMessages}(rs') = \text{acceptedMessages}(rs) + [msg] \wedge\\&\qquad \text{incomingQueue}(rs') = \text{incomingQueue}(rs) - \{msg\}\end{aligned}$$

The system *processes the message* only if the message is allowed to be processed. Otherwise the effect of the message is not visible on the receiving replica. If the message is allowed to be processed, the operation of a data message is added to the data state and the administrative operation of an administrative message if added to the set of access control policies.

$$\begin{aligned}&\text{processMessage}(msg_D, rs, rs') \equiv \text{processOK}(rs, msg_D) \implies\\&\quad \text{persistentOps}(rs') = \text{persistentOps}(rs) + [op_D]\end{aligned}$$

$$\begin{aligned}&\text{processMessage}(msg_{AC}, rs, rs') \equiv \text{processOK}(rs, msg_{AC}) \implies\\&\quad \text{acPolicies}(rs') = \text{acPolicies}(rs) \cup \{op_{AC}\}\end{aligned}$$

A message msg may only be sent by a replica r with replica state rs if

$$\begin{aligned}&\text{msgSender}(msg) = r \wedge\\&\text{sendOK}(rs, msg) \wedge\\&\text{broadcast}(gs, r, msg) \wedge\\&\text{acceptedMessages}(rs') = \text{acceptedMessages}(rs) + [msg] \wedge\\&\text{incomingQueue}(rs') = \text{incomingQueue}(rs) \wedge\\&\text{processMessage}(msg, rs, rs') \wedge\\&\text{operationDeps}(gs', msg) = \text{set}(\text{acceptedMessages}(rs))\end{aligned}$$

Replica r has to be registered as the sender of the message, so the message has the form $msg = (r, op)$ where op is either a data operation or an administrative operation. Message may only be sent and therefore operations on the replica only be performed if they are allowed by the local policies $\text{sendOK}(rs, msg)$. If the permission is granted, the message is locally accepted and processed and broadcast to the other replicas. *Broadcasting* in our case means to add the message to the incoming queue of all other replicas. The operation dependencies of the message are registered to be all the messages that have been accepted by the replica at the time of sending the message.

5.2 Properties of the Data Store

Next, we want to show that the system we have described using the above model is causally consistent. For this, we first show an intermediate lemma.

Lemma 1. *If the global state gs is causally consistent, message msg is causally ready on replica r, r accepts the message and the resulting state is gs', then gs' is also causally consistent.*

Proof. Because gs is causally consistent, we know that for each message m that has been accepted by each replica, the dependencies of m have already been accepted before m. Only r accepts a new message and for all other replicas the list of accepted messages stays the same and the operation dependencies are not changed between gs and gs'. So, regarding the other replicas that systems stays causally consistent. Accepting msg on r means appending msg to the end of the list accepted messages of r

$$\begin{aligned}&\text{acceptedMessages}(replicaState(gs', r)) = \\ &\quad \text{acceptedMessages}(\text{replicaState}(gs, r)) + [msg]\end{aligned}$$

For the prefix of the list that consists of the accepted messages in gs, we known, that all dependencies have already been accepted in this sublist. It remains to show that the newly added message msg does not break the causal consistency. Because we known that msg is causally ready, we also known that the operation dependencies of msg have already been accepted by r, which is the definition of causally ready. This also means that each message in the set of operation dependencies can be found in the list of accepted messages of r, so adding msg to the end of the list of accepted message does also not break causal consistency. Thus we can follow that gs' is also causal consistent. □

Next we can use Lemma 1 to show that steps on the system will not break causal consistency.

Lemma 2 (Causal Consistency Preservation). *If we start in a causal consistent state gs and do a step $gs \longrightarrow gs'$, then gs' is also causal consistent.*

Proof. Case 1 ($gs \xrightarrow{accept} gs'$ accept a message msg on replica r). We assumed gs to be causal consistent and we known from the definition of accepting a message that this message needs to be causally ready. It follows from Lemma 1 that accepting a causally ready message msg in a causal consistent state gs yields a causal consistent state gs'.

Case 2 ($gs \xrightarrow{send} gs'$ send a message msg by replica r). The operation dependencies are changed in gs', the dependencies of msg are set to all messages already accepted by r. So we would have to recheck all accepted messages for all replicas. But we known that all message can be uniquely identified and that msg is a fresh message. This means that none of the replicas have already accepted msg before. In addition, we known that the other replicas except for r do not directly accept msg, but instead have msg in their incoming queue first. From this, we can deduce that causal consistency cannot be broken for the replicas other than r.

In case of r we know, that the replica accepts msg which means we append msg to the end of the accepted messages of r.

$$\begin{aligned}&\text{acceptedMessages}(\mathit{replicaState}(gs', r)) = \\ &\quad \text{acceptedMessages}(\text{replicaState}(gs, r)) + [msg]\end{aligned}$$

We know that msg can not be the prefix of the accepted message that is equal to the previously accepted messages because it is a fresh message. Thus this prefix cannot break causal consistency. It is left to show that appending msg does not break causal consistency. We know that the operation dependencies of msg are set to the messages currectly accepted by r, acceptedMessages(replicaState(gs, r)). Now we have to show that each of these messages have already been accepted by r in gs, which is trivial to see. Thus we have shown that gs' is causal consistent. □

From Lemma 2 we can deduce the correctness of the whole system with respect to causal consistency.

Theorem 1. *The data store as described in our model is causally consistent.*

Proof. By induction on the evaluation steps. □

The last important concept is that of the messages known to a replica. For the list of messages accepted by a replica we can show that all messages are accepted only once per replica. This means we can treat the list of accepted messages as a set. This reinterpretation is done by the set(...) operator. The incomingQueue is already treated as a set to model non-determinism of message transport. The set of *messages known to a replica* is the set of messages accepted by the replica plus the set of messages in the incoming queue

$$\text{knownMessages}(ls) \equiv \text{set}(\text{acceptedMessages}(ls)) \cup \text{incomingQueue}(ls)$$

We can show that all replicas know the same messages by induction over the steps performed.

Lemma 3. *If all replicas known the same messages in gs and we do a step from gs to gs' $gs \to gs'$, then all replicas know the same messages in gs':*

$$\begin{aligned}\forall r1, r2 \in \text{replicas}(gs).\, &\text{knownMessages}(\text{replicaState}(gs, r1)) = \\ &\text{knownMessages}(\text{replicaState}(gs, r2)) \wedge \\ gs \to gs' \implies& \\ \forall r1, r2 \in \text{replicas}(gs').\, &\text{knownMessages}(\text{replicaState}(gs', r1)) = \\ &\text{knownMessages}(\text{replicaState}(gs', r2))\end{aligned}$$

Proof. We again distinguish which kind of step can be made

Case 1 ($gs \xrightarrow{accept} gs'$ accept a message msg on replica r). The accepting replica r removes msg from the incoming queue and adds it to the accepted messages. The known messages stay the same. The known messages of the other replicas also stay the same since their state does not change. □

Case 2 ($gs \xrightarrow{send} gs'$ send a message msg on replica r). The messages msg is directly processed by r and added to the accepted messages. The other replicas get msg in their incoming queue. Overall, msg is added to the known messages of all replicas and therefore the known messages of all replicas are the same. □

5.3 Properties of the Access Control System

Based on the model of the data store we construct the model of the access control system. We show some interesting properties of the access control policies before proving the convergence of the access control state on all replicas.

The access control state is monotonically increasing, old policies are not removed and changes are done by adding the changed policies to the access control state. This can be seen by looking at the possible steps and how messages are accepted by replicas. These changed policies are considered before the old ones when looking for the relevantPolicy.

Using this monotonicity of the access control state, we can show that all known messages can be processed by the sender of the message.

Lemma 4. *All known messages $msg \in \text{knownMessages}(gs)$ are allowed to be processed by the original sender of the message.*

Proof. We show the property by induction over the steps starting in the initial state is.

Case 1 (Initial state). In the initial state none of the replicas have accepted any messages and the incoming queues are empty. This means that there are no known messages yet, so the property hold trivially.

The proofs for the steps can be split-up into two cases, accepting a message and sending a new message:

We assume state gs can be reached from the initial state is by arbitrarily many steps $is \rightarrow^* gs$ and that all known messages in gs can be processed by the sender on the message

$$\forall m \in \text{knownMessages}(gs).\, r = \text{sender}(m) \implies \text{processOK}(r, m)$$

Case 2 ($gs \xrightarrow{accept} gs'$ accept a message msg on replica r). We have to show that all known messages can be accepted by their sender in gs' after processing msg on r. By using that the access control state is monotonically increasing we know that the access control policies in gs are a subset of the access control policies in gs'. We have already seen in the proof of Lemma 3 that the set of known messages does not change during accepting a message. Being allowed to process a message means that the allowing policy is in the access control policies of the replica trying to process the message, in our case the original sender of the message. Since the known messages are the same and the access control policies of gs are a subset of the access control policies of gs' for all replicas that means that the sender is still allowed to process the message.

Case 3 ($gs \xrightarrow{send} gs'$ send a message msg by replica r). In this case have to distinguish the previously known messages knownMessages(gs) and msg, which is added as a new message knownMessages(gs') = knownMessages(gs) $\cup$ $\{msg\}$. We can use the same reasoning as in the accepting case to show that knownMessages(gs) can be processed by their sender in gs'. What is left to show is that msg can be processed by its sender. We know that the sender of msg is r and that msg is allowed to be sent by r. Hence, the allowing policy is the relevant policy in the access control policies of r for the operation performed, which means that the allowing policy is in the access control policies of r. Thus, the known messages in gs' are all allowed to be processed by their sender. □

Using the causal consistency of the data store, we can transfer the property of being able to accept the message to the receiver.

Theorem 2. *All messages are allowed to be processed by the receiving replica once the message is causally ready.*

Proof (sketch). The proof uses Lemma 4 which states that the sender is allowed to process the message. This also means that the policy allowing to process the message is available on the sending replica and therefore gets registered as a dependency of the message to be sent. When another replica r wants to accept message msg, it has to wait until the message is causally ready. Thus all dependencies of msg have been accepted by the replica before accepting msg itself. This makes sure that the message msg_{AC} carrying the policy change that allowed sending the msg has been accepted by r before accepting msg. We can show that the state of the replica is valid, meaning all accepted messages have also been processed by the replica and the effects of the messages have been materialized in the state. Because in a valid state the message msg_{AC} has been processed before processing msg the allowing policy is part of the access control policies of r. This policy then allows processing msg so all messages are allowed to be processed by the receiving replica.

Theorem 2 can be used to show the convergence of the access control state. For the state to converge, no new messages may be sent and the pending messages have to be accepted meaning a system converges in a state where all known messages have been accepted. In Theorem 2 we have shown that all accepted messages are processed by the replicas. This in combination with the set-semantics of the access control state *leads to convergence.*

6 Conclusion

In this paper, we presented the design-space of access control systems for weakly consistent data stores. We showed a formal model of an access control system for causally consistent data stores. One of the main results is that the causally between data operations and access control operations is important for the correctness of the access control system. In addition, we have shown that an applicative model in contrast to the optimistic models proposed by Cherif et al. [6] and Samarati et al. [10] still works without undoing processed operations and still the policies of all replicas eventually converge to a common state.

These results are still rather theoretical and abstract. The next steps will be to develop an access control model inspired by popular models like role-based access control [7, 11] or an authorization logic [2, 3] based on the lower-level model we presented. An implementation of such as system will be based on Antidote [1], a causal-consistent data store developed by the SyncFree Project [2].

[2] https://syncfree.lip6.fr/

References

[1] SyncFree/antidote (Jul 2015), https://github.com/SyncFree/antidote

[2] Abadi, M.: Logic in access control. In: 18th Annual IEEE Symposium of Logic in Computer Science, 2003. Proceedings. pp. 228–233. IEEE Comput. Soc (2003)

[3] Bauer, L.: Access Control for the Web via Proof-Carrying Authorization. Ph.D. thesis, Princeton University (2003)

[4] Bieniusa, A., Zawirski, M., Preguiça, N.M., Shapiro, M.: An optimized conflict-free replicated set. arXiv.org (2012)

[5] Burckhardt, S.: Principles of Eventual Consistency. Foundations and Trends in Programming Languages 1(1-2), 1–150 (2014)

[6] Cherif, A., Imine, A., Rusinowitch, M.: Practical access control management for distributed collaborative editors. Pervasive and Mobile Computing 15, 62–86 (2014)

[7] Ferraiolo, D., Kuhn, R.: Role-Based Access Control. In: In 15th NIST-NCSC National Computer Security Conference. pp. 554–563 (1992)

[8] Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News 33(2), 51 (2002)

[9] Preguica, N., Zawirski, M., Bieniusa, A., Duarte, S., Balegas, V., Baquero, C., Shapiro, M.: SwiftCloud: Fault-Tolerant Geo-Replication Integrated all the Way to the Client Machine. In: 2014 IEEE 33rd International Symposium on Reliable Distributed Systems Workshops. vol. abs/1310.3, pp. 30–33. IEEE (Oct 2014)

[10] Samarati, P., Ammann, P., Jajodia, S.: Maintaining Replicated Authorizations in Distributed Database Systems. Data Knowl. Eng. 18(1), 55–84 (1996)

[11] Sandhu, R.S., Coyne, E.J., Feinstein, H.L., Youman, C.E.: Role-Based Access Control Models. IEEE Computer 29(2), 38–47 (1996)

[12] Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506 (Jan 2011)

[13] Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-free Replicated Data Types. Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems pp. 386–400 (2011)

[14] Wobber, T., Rodeheffer, T.L., Terry, D.B.: Policy-based access control for weakly consistent replication. In: Morin, C., Muller, G. (eds.) European Conference on Computer Systems, Proceedings of the 5th European conference on Computer systems, EuroSys 2010, Paris, France, April 13-16, 2010. pp. 293–306. ACM (2010)

Terminierungsanalyse in Service-Orientierten Systemen

Mandy Weißbach and Wolf Zimmermann

Martin-Luther-University Halle-Wittenberg, Institute of Computer Science

Abstract. Ein großer Teil der zur Zeit verwendeten Softwaresysteme sind nach dem Prinzip der Service-orientierten Architektur aufgebaut. Diese Softwaresysteme bestehen aus einzelnen Services oder Diensten, die mehr oder weniger stark miteinander kommunizieren, um so zum Beispiel einen komplexen Geschäftsprozess realisieren können. In unserer Arbeit zeigen wir anhand eines Beispiels, dass durch die Kommunikation der Services das Terminierungsverhalten des Systems beeinträchtigt werden kann. Wir identifizieren diese Schwachstellen und stellen Anforderungen bezüglich der Terminierungsanalyse in Service-orientierten Systemen auf. Weiterhin betrachten wir bestehende Methoden zur Terminierungsanalyse und bewerten den möglichen Einsatz dieser Analysen in Service-orientierten Systemen und stellen erste Lösungsansätze vor.

Typprüfung und Namensanalyse für textuelle Anforderungsbeschreibungen[*]

Sebastian Wendt und Wolf Zimmermann

Martin-Luther-Universität Halle-Wittenberg
sebastian.wendt@informatik.uni-halle.de
wolf.zimmermann@informatik.uni-halle.de

Zusammenfassung. Anforderungsbeschreibungen für eine technische Domäne sollen in einer Sprache verfasst werden, die Mehrdeutigkeiten eliminiert, und die automatisch durch einen Übersetzer überprüft werden kann. Dieser Übersetzer soll die konsistente Verwendung von Namen mittels Namensanalyse und die korrekte Kombination von in einem Glossar definierten Begriffen mittels Typanalyse überprüfen.

1 Einleitung

Natürliche Sprache ist in vielen Projekten der Ausgangspunkt zur Modellierung von Anforderungen. Um ein Anforderungsmanagement umzusetzen, das alle Phasen eines Projekts umfasst, müssen diese natürlichsprachlichen Anforderungen durch die Werkzeuge mit einbezogen werden. Rückverfolgbarkeit von der Implementierung zurück, nicht nur bis zur ersten formalen Beschreibung, sondern bis zur textuellen Beschreibung, kann dann auch Anforderungen von Stakeholdern erfassen, die nicht mit formalen Modellbeschreibungen umgehen können. Wenn natürliche Sprache von automatisierten Werkzeugen behandelt wird, dann gewinnt der Prozess zudem an Geschwindigkeit, Zuverlässigkeit und Reproduzierbarkeit.

Für die Verarbeitung von natürlichsprachlichen Texten wird oft auf heuristische und probabilistische Techniken zurückgegriffen. Probabilistische Techniken versprechen, die Analyse eines Textes dann zuverlässig durchzuführen, wenn man sie auf einem hinreichend großen Datensatz (Korpus) trainiert. Doch wie sich Stakeholder nicht über Begrifflichkeiten einigen können, so gibt es auch Abweichungen zwischen den Annotationen in den Korpora und den Autoren der Anforderungsdokumente einer technischen Domäne. Je umfangreicher der Korpus ist, desto wahrscheinlicher ist es, auf Interpretationen zu treffen, die für die Domäne nicht relevant sind und unnötige Mehrdeutigkeiten einführen, die Fehler verschleiern. Eine hoher Anteil an falsch-positiven Interpretationen (Falsches, das als richtig erkannt wird) ist die Folge.

Wir beschreiben in Abschnitt 3 die konkrete Syntax einer kontrollierten Sprache, welche die Syntax natürlicher Sprache (Deutsch) imitiert. Die Syntax basiert

[*] Diese Arbeit wurde im Rahmen des vom Bundesministerium für Bildung und Forschung (BMBF) geförderten Forschungsprojekts „ELSY“ (Nr. 16M3202D) erstellt.

auf der Idee der Schablonen, und setzt sie als Grammatik um, statt es bei einer Menge von *best-practice* Formulierungsrichtlinien zu belassen. Aus dieser Grammatik erzeugen wir einen Parser, der Teil eines Übersetzers ist.

In den Abschnitten 4 und 5 beschreiben wir Namensanalyse und Typprüfung, die dazu dienen, die Konsistenz innerhalb der Anforderungsbeschreibung abzusichern.

Die Ziele dieser Arbeit sind:

- Eindeutigkeit der textuellen Formulierung herzustellen und Mehrdeutigkeit natürlicher Sprache zu eliminieren,
- Überprüfbarkeit der Beschreibung herzustellen, vollständig und automatisch, statt heuristisch oder per Hand,
- Konsistenz der Beschreibungen sicherzustellen, durch Typanalyse statt *best-practice* Regeln
- Lesbarkeit der Texte zu erhalten, statt eine formale Syntax zu benutzen, die Stakeholder ohne Einarbeitung ausschließt.

2 Verwandte Arbeiten

Eine kontrollierte Sprache, wie wir sie in dieser Arbeit vorstellen, ist im Allgemeinen eine natürliche Sprache, der gewisse syntaktische und semantische Einschränkungen auferlegt sind. Dadurch wird eine kontrollierte Sprache noch nicht zu einer formalen Sprache. So sind *Schablonen* [Hull u. a., 2005; Rupp, 2014] Vorlagen für den organisierten Satzaufbau, um Anforderungstexte zu schreiben, die für Dritte leichter lesbar sind.

Leichtes Deutsch wird von öffentlichen Einrichtungen auf deren Webseiten eingesetzt, um Inhalte barrierefrei für Personen mit Verständnisschwierigkeiten zu transportieren. *Einfache Sprache* verfolgt ebenfalls dieses Ziel und ist für Nicht-Deutsch-Muttersprachler entwickelt worden.

Rechtschreibprüfungs-Programme als Bestandteil von Textverarbeitungs-Programmen vergleichen im einfachsten Fall Wörter mit einem Wörterbuch. Es gibt weiter auch solche, dich auch die Grammatik überprüfen, z.B. auf Basis von *link grammars* von Lachowicz u. a. [2015]. Rechtschreibprüfungsprogramme dürfen nicht zu viele Fehlermeldungen produzieren, da Nutzer das Programm sonst schnell als zu empfindlich und untauglich abtun. Daher zeigen sie weniger potentielle Fehler an (mehr falsch-Positive, weniger Negative).

Die Prädikat-Argument-Struktur der Präpositionalphrasen in konkreter Syntax ist der Prädikatenlogik bzw. Prolog nachempfunden. Kuhn u. Bergel [2014] zeigten in einem Experiment an Studentengruppen, dass Gruppen, die Aussagen in natürlicher Sprache verarbeiten sollten, genauer und schneller arbeiteten, als Gruppen, die dieselben Aussagen in Form von Prolog-Statements erhielten. Die Prädikat-Argument-Struktur ist auch Grundlage der konstruierten natürlichen Sprache Lojban Cowan [2000].

CIRCE [Ambriola u. Gervasi, 2006] ist ein Übersetzer, der Anforderungsbeschreibungen in natürlicher Sprache (Italienisch) auf heuristische Art analysiert und in der Lage ist, Transformationen nach UML, ER-Diagramm oder

ASM durchzuführen. Die Grammatik des Eingabetextes ist nicht beschränkt; der Übersetzer unterzieht den Text einem *part-of-speech-tagging*, und führt dann iterativ eine Bewertung einer Menge von Regeln auf den klassifizierten Token durch, woraus sich eine Punktwertung für jede mögliche Regelanwendung ergibt, wendet die am besten bewertete Regel an, und wiederholt den Vorgang bis ein Terminierungskriterium erreicht ist. Die Regeln beziehen sich auf Attribute (*tags*) der Token, die aus der syntaktischen Analyse oder aus Regelanwendungen stammen können. Beispiele zeigen Regeln, die ein Typsystem beschreiben. Die Regeln können durch die Benutzer verändert werden und sind in einer speziellen formalen Syntax verfasst.

[Farfeleder, 2012] beschreibt in seiner Dissertation ein Verfahren zur Transformation von natürlicher Sprache in Schablonen auf Englisch (*boilerplates*) für die Domäne der eingebetteten Systeme. Ausgangspunkt sind vorliegende Texte, die durch halbautomatische, überwachte Transformation in eine Form gebracht werden soll, die besser analysierbar ist, d.h. deren Analyse weniger korrigierende Eingriffe benötigt. Die Schablonen werden benutzt um daraus Informationen über die Sicherheit des modellierten Systems zu gewinnen (*safety analysis*).

3 Kontrollierte natürliche Sprache

In dieser Arbeit entwickeln wir eine Sprache, die das Aussehen natürlicher Sprache besitzt, und wählen die syntaktischen Einschränkungen so, dass wir eine formale Sprache erhalten. Dadurch bleibt die Lesbarkeit für Dritte ohne Vorkenntnisse erhalten. Auf dieser Sprache führen wir in späteren Abschnitten Konsistenzprüfungen mittels Namensanalyse und Typprüfung durch und definieren zu diesem Zweck ein domänenspezifisches Glossar. Die größte Schwierigkeit bei Projekten, die natürliche Sprache verarbeiten (*natural language processing*, NLP) besteht in dem Anspruch der Nutzer, das Chaos an überladenen Bedeutungen und Sonderfällen, das wir als natürliche Sprache bezeichnen, mit gleicher oder besserer Präzision und Vorhersagekraft verarbeiten zu können. Diese Arbeit erhebt nicht den Anspruch, natürliche Sprache vollständig abzudecken. Der Anteil an natürlichsprachlichen Ausdrücken, der von unseren Konsistenzprüfungen abgelehnt wird (falsch-Negative) ist daher groß im Vergleich zu NLP-Parsern. Im Gegenzug streben wir an, den Anteil an Ausdrücken, der nicht korrekt ist, und der trotzdem akzeptiert wird (falsch-Positive) zu eliminieren . Dies gelingt uns durch die Einschränkungen, die wir treffen[1].

Die Merkmale unserer kontrollierten Sprache, auf die wir anschließend im Einzelnen eingehen werden, sind:

- Prädikat-Argument-Struktur wie in Prädikatenlogik (Abschnitt 3.1)
- Unterscheidung Definition und Benutzung durch Artikel (*determiner*) (Abschnitt 4)
- Bedingte Anforderungen (Wenn-Dann) (Abschnitt 3.2)
- Zusammenfassen der Zeitformen (Abschnitt 3.3)

[1] zu beachten sind einige Spezialfälle; siehe Abschnitt 3.4

Morphologie verwerfen Eine Grundannahme der konkreten Syntax ist, dass wir, wo möglich, auf morphologische Informationen (Beugung, Deklination) verzichten wollen. Diese Informationen sind inhärent unzuverlässig, weil durch die Entwicklung der Sprache viele Fälle entstanden sind, in denen syntaktisch identische Terme für semantisch unterschiedliche Konzepte verwendet werden müssen. Dies gilt für Artikel (der Mann, Nominativ; der Frau, Genitiv), Substantive (Anzeige, wie Bildschirm; Anzeige, wie Gerundium zu *anzeigen*) und Verben (schnellen, wie *hervorschnellen*; schnellen wie Partizip 2 Plural von *schnell*). Morphologische Information erlaubt im Allgemeinen keine eindeutige Klassifizierung. Wir verwerfen daher Informationen über Singular oder Plural, grammatisches Geschlecht und Deklination von Substantiven, Adjektiven und Artikeln (Fälle). Die Konjugation von Verben klassifizieren wir in 2 Klassen entsprechend der Zeitform und verwerfen die Information über Zahl und Person.

3.1 Prädikat-Argument-Struktur

Unbedingte Anforderungssätze bilden den Grundbaustein für Schablonen in der konkreten Grammatik. Die Struktur der Anforderungssätze, die wir in unserer kontrollierten Sprache erlauben, ähnelt der von Prädikaten in Prädikatenlogik. Für Benutzer ohne Kenntnis von Prädikatenlogik ist dies jedoch nicht zu erkennen und nicht erforderlich.

Ein Anforderungssatz in kontrollierter Sprache könnte wie im folgenden Beispiel lauten:

> *Der Benutzer „Anlagenfahrer“ soll das System „Pumpe“ über das Bedienterminal innerhalb 1 Minute hochfahren können.*

Die konkrete Grammatik dieses Satzes besteht aus einer Verbalphrase (VP) mit mehreren Präpositionalphrasen (PP):

```
VP ::= ( PP | Modalverb )* Prädikat Modalverb? .
PP ::= Präposition? Artikel? Prädikat Bezeichner? .
```

Präposition, Artikel und Bezeichner sind optional. Die Bestandteile des Beispielsatzes lassen sich wie in Tabelle 1 aufteilen.

	Präposition	Artikel	Prädikat	Bezeichner	Modalverb
PP		Der	Benutzer	Anlagenfahrer	
Modal					soll
PP		das	System	Pumpe	
PP	über	das	Bedienterminal		
PP	innerhalb	1	Minute		
Prädikat			hochfahren		
Modal					können

Tabelle 1. Beispiel Prädikat-Argument-Struktur, zeilenweise zu lesen

In der Praxis sind Sätze meist kürzer und enthalten weniger Präpositionen. Im Anhang befindet sich ein Ausschnitt aus einem technischen Handbuch und seine Übersetzung in unsere Sprache.

Präpositionen Präpositionen sind eine endliche Mengen von Wörtern (*closed set*). Wir setzen sie ein als Beschriftung (*label*), um die Argumente eines Prädikats einer semantischen Rolle, d.h. einem der Parameter des Prädikats zuzuordnen. Dieses sogenannte *semantic role labeling* ist ein Teilproblem von NLP und wird i.d.R. heuristisch bzw. probabilistisch gelöst (Palmer u. a. [2010]). Wir tragen der Beweglichkeit in der natürlichen Sprache dadurch Rechnung, dass wir das freie Verschieben von Argumenten mit unterschiedlichen Beschriftungen erlauben, fordern aber, dass Argumente mit der gleichen Beschriftung der Reihenfolge folgen, wie sie für das jeweilige Prädikat im Glossar definiert wurde. Diese Regelung betrifft am häufigsten die *leere Beschriftung* (ε), die für Argumente ohne Präposition angenommen wird.

Beispiel[2]:

(1) *Der Benutzer soll die Pumpe mit einem Fördermedium füllen.*
(2) *Der Benutzer soll mit einem Fördermedium die Pumpe füllen.*
(3) *Die Pumpe mit einem Fördermedium soll der Benutzer füllen.*

In Prädikatschreibweise:

(1) füllen(Benutzer,Pumpe,~mit:Fördermedium)
(2) füllen(Benutzer,~mit:Fördermedium,Pumpe)
(3) füllen(Pumpe,~mit:Fördermedium,Benutzer)

Die Sätze (1) und (2) erzeugen eine äquivalente Prädikatstruktur, da die Reihenfolge von *Benutzer* und *Pumpe* erhalten bleibt. Bei Satz (3) ist dies nicht der Fall.

Präpositionen sind gelegentlich Kontraktionen unterworfen, z.B. wird *von+dem* zu *vom*. Wir lösen diese Kontraktionen vor der Typprüfung auf, so dass Verbdefinitionen nicht auf kontrahierte Varianten Rücksicht nehmen müssen.

Artikel Artikel sind eine endliche Menge von Wörtern. Wir setzen sie zusammen mit Präpositionen als syntaktischen Trenner für die Argumente eines Prädikats in einer Liste von Präpositionalphrasen (PP) ein. Wir verwerfen die Information über den grammatischen Fall (Kasus) aus dem Artikel, da diese nicht zuverlässig ist. Zulässige Artikel bzw. sogenannte *determiner* sind:

- unbestimmte Artikel (ein, eine)
- bestimmte Artikel (der, die, das)
- Kardinalzahlen (1,2,...)
- Negationen (kein, keine)

[2] vgl. erster Satz im Anhang A.2; Prädikat im Aktiv ergänzt um *Benutzer*

Wir benutzen unbestimmte Artikel, um eine definierende Instanz eines Bezeichners zu kennzeichnen, und bestimmte Artikel und Kardinalzahlen, um benutzende Instanzen zu kennzeichnen (siehe auch Abschnitt 4). Negationen wirken als definierende Instanzen.

Prädikate Das Prädikat als Argument deklariert den a-posteriori Typ des darauffolgenden Bezeichners. Existiert kein Bezeichner, so wird ein anonymer Bezeichner angenommen. Im aktuellen Gültigkeitsbereich muss dann genau ein Bezeichner vorhanden sein, der sich an den deklarierten Typ anpassen lässt. Ist kein solcher Bezeichner oder mehrere vorhanden, handelt es sich um einen Typfehler.

Das Prädikat als Verb konstituiert einen typisierten Ausdruck. Der Typ der Argumente (PP des Satzes) wird gegen die Typen der für das jeweilige Literal deklarierten Parameter geprüft (laut Glossar). Die Argumente werden entsprechend ihrer Beschriftungen (*labels*) sortiert; die Reihenfolge gleich beschrifteter Argumente jedoch nicht verändert.

Bezeichner Bezeichner verbinden die Menge von Anforderungen untereinander. Bezeichner sind optional; wenn sie entfallen wird eine anonyme Referenz erzeugt, die einen im Gültigkeitsbereich eindeutigen Typ besitzen muss, über den sie identifiziert wird. Wir unterscheiden definierende und benutzende Instanzen von Bezeichnern und führen eine Namensanalyse durch (siehe Abschnitt 4).

Modalverben Modalverben im Satz beschreiben das Merkmal der Variation für die Anforderung. Wir unterscheiden:

- Soll-Anforderungen
- Kann-Anforderungen
- Darf-Nicht-Anforderungen
- Tun

Die Art des Modalverbs beschreibt die, ob ein Variationspunkt für die betreffende Anforderung erstellt wird: Eine Kann-Anforderung impliziert mehrere Umsetzungsmöglichkeiten, während eine Soll-Anforderung strikt nach genau einer Umsetzung verlangt. Eine Darf-Nicht-Anforderung bezeichnet eine negative Abhängigkeit im Anforderungsgraphen, so dass die bezeichnete Anforderung nicht erfüllt sein darf, wenn die bezeichnende Anforderung erfüllt sein soll.

Das Modalverb *tun* dient behelfsweise bei der Umformulierung von einer Zeitform in eine andere (Abschnitt 3.3). Sätze mit diesem Modalverb werden wie Sätze ohne Modalverb, d.h. Bedingungen, behandelt.

3.1.1 Relativsätze Relativsätze dienen in unserer Sprache der Verkürzung der Beschreibung. Statt zwei getrennte Hauptsätze hintereinander zu schreiben, kann ein Argument eines Satzes gewissermaßen *inline* durch eine Beschreibung in einem Relativsatz näher bestimmt oder näher beschrieben werden. Relativsätze sind syntaktischer Zucker und können zu separaten Hauptsätzen umgewandelt werden. Wir unterscheiden zwei Arten von Relativsätzen:

- beschreibende Relativsätze
- bestimmende Relativsätze

Beschreibende Relativsätze müssen ein Modalverb (kann,soll,..) enthalten. Sie werden in separate Anforderungen übersetzt. Bestimmende Relativsätze sind Relativsätze, die kein Modalverb enthalten; sie werden in eine Bedingung übersetzt. Auf diese Weise kann eine Anforderung zu einer bedingten Anforderung werden, auch ohne die explizite Wenn-Dann-Syntax zu verwenden (Abschnitt 3.2).

Relativsätze sind ein Kompromiss zwischen der Normalform, die nur aus Hauptsätzen besteht, und der gebräuchlichen Schriftsprache, die Bedingungen, Anforderungen und zugehörige (implizierte) Prädikate in ein einziges Partizip zwängt. Eine übliche (aber nach unserer Syntax nicht zulässige) Präpositionalphrase könnte lauten:

Die kapazitiven Lasten

Die PP muss umformuliert werden, in eine der beiden Varianten:

Die Lasten, die kapazitiv sind, ...
Die Lasten, die kapazitiv sein sollen, ..

Die erste Variante, wenn eine bestimmende Wirkung (nur solche Lasten, die kapazitiv sind) beabsichtigt ist, und die zweite, wenn eine beschreibende Wirkung (Lasten, die außerdem immer kapazitiv sein sollen) beabsichtigt ist.

Relativsätze ziehen leider einige Schwierigkeiten bezüglich der Argumentreihenfolge nach sich, die wir in Abschnitt 3.4 betrachten.

3.2 Bedingte Anforderungen

Eine Anforderungsbeschreibung untergliedert sich im Allgemeinen in einen Glossar-Teil, der für alle Projekte innerhalb einer Domäne (eines Gewerks) gültig ist und vorab erstellt wird, sowie einen Anforderungs-Teil, der projektspezifisch ist.

Eine Anforderung besteht im Allgemeinen aus einer Bedingung, unter der sie zu erfüllen ist, und einer Konsequenz, der beschreibt, was zu erfüllen ist:

```
Anforderung ::= (Bedingungen 'dann')? Konsequenz .
Bedingungen ::= Bedingungen 'und' 'wenn' VP | 'Wenn' VP .
Konsequenz  ::= VP .
Anforderung ::= Bedingungen 'dann' ':' Spiegelstriche .
Spiegelstriche ::= Spiegelstriche '-' VP | '-' VP .
```

Wird keine Bedingung angegeben, so soll die Konsequenz universell gültig sein.

Bedingungen (Wenn) beschreiben eine zur Laufzeit erfolgte Zustandsänderung („ist gestartet“), oder eine Eigenschaft (Invariante, „ist aus Holz“), oder eine Modell-Variante („kann aus Holz sein“). In der konkreten Syntax können sie statt eines „echten“ Modalverbs auch den Platzhalter „tun“ enthalten.

Konsequenzen (Dann) beschreiben eine Anforderung und deren Modus (*soll/muss*, *kann/muss-nicht*, *darf-nicht*). In der konkreten Syntax müssen sie immer ein Modalverb enthalten.

Durch eine Art von Relativsätzen – bestimmende Relativsätze – können Anforderungen um eine Bedingung erweitert werden, ohne dass die charakteristischen Schlüsselwörter *wenn* und *dann* vorhanden sind (siehe Abschnitt 3.1.1). Dies ist ein Kompromiss zu Lasten der einfachen Lesbarkeit und zu Gunsten der leichten Transformierbarkeit bestehender Anforderungen (mit Nebensätzen) in solche der kontrollierten Sprache.

Ein Erweiterung, von der wir im Beispiel im Anhang Gebrauch machen, sind Spiegelstriche anstatt der Konsequenzen, die vermeiden, dass die Bedingung in jedem Satz wiederholt werden muss.

3.3 Zeitformen

Für die Beschreibung von Systemverhalten erachten wir eine Beschreibung des Zustands zeitlich relativ zu Operationen, die auf ihn einwirken können, für erforderlich. Wir wollen daher in 3 Zeitformen unterscheiden:

Vorzeitigkeit $<$ Die durch das Verb angegebene Handlung hat noch nicht stattgefunden (entspricht Futur 1).

Gleichzeitigkeit $=$ Die Handlung findet gerade statt (entspricht Partizip 1).

Nachzeitigkeit $>$ Die Handlung ist abgeschlossen (entspricht Perfekt).

Durch Kombination von Zeitformen und Passiv ergibt sich eine Vielzahl von gebeugten Verbformen. Die gebeugten Verbformen sollen auf eine Normalform reduziert werden. Die Tabelle zeigt die Kombinationsmöglichkeiten von Zeitform und Aktiv/Passiv. Die Bestimmung der Zeitform ist nicht heuristisch, nur unübersichtlich.

Die Tabelle 2 unterstrichenen Varianten sind verboten, denn:

$=_{[d]}$: Aussagesätze sind keine Anforderungen
$<_{[d]}$: Anforderungen dürfen nicht rückwirkend formuliert sein
† Vertauschen von unmarkierten Argumenten ohne Passiv verändert die Semantik ohne die Syntax anzupassen (Artikel, sog. *determiner* werden verworfen)

Anhand der Information über das Modalverb (*soll*, *kann*, *tun* oder keines) lässt sich aus dem Hilfsverb (*werden* usw.) die Zeitform eindeutig bestimmen, wie in Tabelle 3 dargestellt.

Aus den normalisierten Zeitformen und dem Wenn-Dann-Kontext lässt sich die folgende Semantik ableiten:

Zeitform	Aktiv	Passiv
Anforderung ($<_{[d]}$)	Der A soll das B dem C geben.	Das B soll dem C *durch* A gegeben werden.
Anforderung ($=_{[d]}$)	Der A gibt das B dem C.	Das B wird dem C durch A gegeben.
Anforderung ($>_{[d]}$)	Der A soll das B dem C gegeben haben.	Das B soll dem C durch A gegeben worden sein.
Zukunft ($<_{[w]}$)	(Wenn) der A das B dem C geben (soll)	(Wenn) das B dem C durch A gegeben werden (soll)
Gegenwart ($=_{[w]}$)	(Wenn) der A dem B das C gibt (~~soll~~)	(Wenn) das B dem C durch A gegeben wird
Perfekt ($>_{[w]}$)	(Wenn) der A dem B das C gegeben hat	(Wenn) das B dem C durch A gegeben wurde/worden ist

Verb ohne unmarkiertes Akkusativobjekt: keine Passivform vorhanden

Anforderung ($<_{[d]}$)	Das A soll dem B gleichen.	†Dem B soll das A gleichen. Dem A soll das B gleichen(d) werden.
Zukunft ($<_{[w]}$)	(Wenn) das A dem B gleichen (soll)	†(Wenn) dem B das A gleichen (soll) (Wenn) dem A das B gleichen(d) werden (soll)
Gegenwart ($=_{[w]}$)	(Wenn) das A dem B gleicht (~~soll~~)	†(Wenn) dem B das A gleicht (Wenn) das A dem B gleichend ist
Perfekt ($>_{[w]}$)	(Wenn) das A dem B geglichen hat	†(Wenn) dem B das A geglichen hat (Wenn) das A dem B gleichend gewesen ist

Adjektiv-Prädikat (ohne Objekt):

Anforderung ($<_{[d]}$)	Das A soll aktiv sein.	Das A soll aktiv werden.
Zukunft ($<_{[w]}$)	(Wenn) das A aktiv sein soll (soll)	(Wenn) das A aktiv werden (soll)
Gegenwart ($=_{[w]}$)	(Wenn) das A aktiv ist (~~soll~~)	(Wenn) das A aktiv geworden ist
Perfekt ($>_{[w]}$)	(Wenn) das A aktiv war	(Wenn) das A aktiv gewesen ist

Tabelle 2. Zeitformen in Kombination mit Passiv-Aktiv-Formen

Zeitform	Aktiv	Passiv
Anforderung ($<_{[d]}$)	soll + Infinitiv	soll + Partizip + werden
Anforderung ($=_{[d]}$)	Präsens	werden + Partizip
Anforderung ($>_{[d]}$)	Partizip + haben.	Partizip + worden sein.
Zukunft ($<_{[w]}$)	(Wenn) Infinitiv (soll)	(Wenn) Partizip + werden (soll)
Gegenwart ($=_{[w]}$)	(Wenn) Präsens (~~soll~~) / (Wenn) Infinitiv + tun (~~soll~~)	(Wenn) Partizip + werden (~~soll~~)
Perfekt ($>_{[w]}$)	(Wenn) Partizip + haben	(Wenn) Partizip + wurden/worden sind

Tabelle 3. Zusammenfassung Zeitformen

- Anforderung ($<_{[d]}$): Definition einer Nachbedingung
- Zukunft ($<_{[w]}$): Introspektion (*reflection*) einer Anforderung
- Gegenwart ($=_{[w]}$): Prüfen einer Invariante (über einen Zeitverlauf)
- Perfekt ($>_{[w]}$): Auslösen eines Ereignisses/Signals/Callbacks

3.4 Spezialfälle der konkreten Syntax

Obwohl die konkrete Syntax so einfach wie möglich gehalten wurde, kann sie nicht alle mehrdeutigen Formulierungen natürlicher Sprache unterbinden. Relativpronomen sind im Allgemeinen nicht geeignet, um die beabsichtigte Position eines Arguments in der Parameterliste festzustellen, wie das folgende Beispiel zeigt:

Die Pumpe, die das System steuert.

Soll hier die Pumpe das (Fluid-)System steuern, oder durch das (Steuer-)System gesteuert werden? Da das Relativpronomen fest an erster Stelle des Relativsatzes steht, sind Konventionen notwendig, um die korrekte Reihenfolge der Argumente herzustellen.

In diesem Abschnitt stellten wir die konkrete Syntax einer kontrollierten Sprache vor, die Lesbarkeit von natürlicher Sprache erhält und die Voraussetzungen für Überprüfbarkeit wie eine formale Sprache bereitstellt.

4 Namensanalyse

Wir wollen eine Namensanalyse durchführen, um sicherzustellen, dass Objekte, die bei der Beschreibung von Prozessen herangezogen werden, zuvor selbst beschrieben wurden. Diese Vollständigkeitsprüfung ist Teil unseres Ziels, natürliche Sprache automatisiert überprüfbar zu machen. Die im vorangegangenen Abschnitt vorgestellt Syntax liefert die Grundlage dazu.

Um eine Namensanalyse auf Ebene der Bezeichner, die Bestandteil der Präpositionalphrasen (PP) sind, müssen wir unterscheiden zwischen definierenden und benutzenden Bezeichnern. Wir erlauben kein Verdecken (*shadowing*) von Bezeichnern; mehrmalige Definitionen eines Bezeichners mit dem gleichen Namen sind Fehler. Den Bezeichnern wird bei ihrer Definition der Typ des vorangestellten Prädikats zugewiesen.

Wir unterscheiden zwischen Definition und Benutzung anhand des Artikels:

Definition *Eine Pumpe P1 …*
Benutzung *Die Pumpe P1 …*

Unbestimmte Artikel (ein, eine, usw.) definieren einen Bezeichner, bestimmte Artikel (der, die, das, usw.) und Kardinalzahlen (1,2, usw.) greifen benutzend auf einen Bezeichner zu.

Anonyme Bezeichner Im Sprachgebrauch wird eine Instanz regelmäßig nicht benannt, wenn der Typ ein Objekt eindeutig identifiziert.

Es ist möglich, den Bezeichner in einer Präpositionalphrase auszulassen. Der Übersetzer erzeugt dann eine anonyme Referenz mit dem deklarierten Typ. Auf diese anonyme Referenz kann über den Typ zugegriffen werden, wenn dieser im Gültigkeitsbereich eindeutig ist. Ist ein anonymer Bezeichner nicht eindeutig über den Typ identifizierbar, liegt ein Fehler vor. Eine Ausnahme bilden implizite Bezeichner für Verben im folgenden Abschnitt.

Beispiel[3]:

> *Wenn eine Pumpe in Betrieb genommen werden soll, dann ...*
> *... die Pumpe muss mit einem Fördermedium gefüllt sein.*

Im ersten Beispielsatz wird ein anonymer Bezeicher b_1 für *Pumpe* erzeugt. Im zweiten Satz kann auf diesen anonymen Bezeicher b_1 zugegriffen werden, da b_1 der einzige Bezeichner vom Typ *Pumpe* ist, der außerdem keine Untertypen hat[4].

Verben erzeugen anonyme Bezeichner Unsere Syntax erlaubt keine Adverbien, Adjektive oder andere vorangestellten Modifikatoren. Trotzdem ist es notwendig, Prozesse näher zu beschreiben. In natürlicher Sprache dienen Adverbien diesem Zweck; eine Konvention, um für Verben direkt bei deren Gebrauch mit einen Bezeichner zu benennen, hat sich in natürlicher Sprache nicht herausgebildet. Vermutlich liegt dies daran, dass Menschen eher überladene Verbbedeutungen dynamisch auflösen als für jeden Zweck neue Verben zu kreieren. Im technischen Kontext wollen wir aber genau das tun, um anhand der Literale ihre Bedeutung unterscheiden zu können – und nicht anhand des Kontextes, was immer das auch sein mag.

Wollen wir Verben modifizieren, können wir die nur tun, indem wir sie als Argumente innerhalb eines anderen Satzes benutzen. Für jede Benutzung eines Verbs erzeugt unser Übersetzer implizit einen anonymen Bezeichner. Im Gegensatz zu implizit deklarierten Argumenten, ist es kein Fehler, wenn ein impliziter Bezeichner, der auf ein Verb verweist (erkennbar anhand seines a-priori Typs), nicht eindeutig einer Instanz zugeordnet werden kann. In diesem Fall wird die zuletzt erzeugte Instanz gewählt und eine Warnung ausgegeben. Grund ist, dass syntaktisch keine direkte Möglichkeit besteht, Bezeichner für Verben zu deklarieren.

Beispiel:

> *Die Pumpe muss angeschlossen werden*$_1$.
> *Der Frequenzumrichter muss angeschlossen werden*$_2$.
> *Das Anschließen*$_2$ *(..) muss vorschriftsmäßig sein.*

[3] siehe Abschnitt A.2, erster Absatz

[4] siehe Diagramm der Typhierarchie in Abbildung 2 im Anhang

Das Verb *angeschlossen werden* erzeugt zwei anonyme Instanzen. Dem Verb ist im Glossar der Typ *Anschließen* zugewiesen[5]. Mittels dieses Typs wird auf die (zuletzt) im zweiten Satz deklarierte Instanz zugegriffen. Der Übersetzer erzeugt dazu eine passende Warnung, dass der anonyme Bezeichner mehrere Kandidaten hatte, um auf eventuelle Fehler in der Reihenfolge der Sätze aufmerksam zu machen.

5 Typsystem

Wir wollen eine Typanalyse durchführen, um sicherzustellen, dass gleich benannte Objekte auch die gleiche Sache bezeichnen. Dazu vergleichen wir den deklarierten Typ bei der Definition eines Bezeichners mit dem deklarierten Typ bei dessen Benutzung. Weiterhin vergleichen wir in Abschnitt auch die deklarierten (a-priori) Typen mit den von der Umgebung erwarteten (a-posteriori) Typen. Diese Konsistenzprüfung ist ein weiterer Schritt zu unserem Ziel, natürliche Sprache überprüfbar zu machen.

Das Glossar ist die Sammlung aller Typdeklarationen, die durch den Domäneningenieur geschrieben werden sollen. Das Glossar soll für die Domäne relevanten Klassen von Objekten (Pumpen, Systeme, Nachrichtenarten, Anschlussarten, usw.), von Merkmalen (energieeffizient, schnell, blau, usw.) und von Prozessen (anschalten, benachrichtigen, anschließen, usw.) als Typdeklarationen enthalten. Das Typsystem kann in der Definitionstabelle des Übersetzers abgelegt werden. Die Typdeklarationen unterscheiden sich von und werden ergänzt durch die Definition von Verben, die wir in Abschnitt 5.3 erläutern.

5.1 Typdeklarationen

Typdeklarationen bilden den benutzerdefinierten Teil der abstrakten Syntax der domänenspezifischen Anforderungssprache. Typdeklarationen beschreiben, was man linguistisch als *Konzepte* bezeichnen würde. Sie sind Äquivalenzklassen, die Verben, Adjektive und Substantive zu Prädikaten zusammenfassen. Im nichtbenutzerdefinierten Teil der abstrakten Syntax befinden sich z.B. die Interpretation der Modalverben (soll, kann) aus Abschnitt 3.1. Eine einfache Typdeklaration für ein Substantiv lautet z.B.:

> *Eine Pumpe ist ein System.*
> *Ein System ist ein Objekt*

Diese Deklaration deklariert den Typ *Pumpe* und setzt ihn zugleich in Untertypsbeziehung zum Typ *System*. Der Typ *Objekt* ist vordefiniert und gemeinsamer Obertyp aller Substantive in unserer Sprache. Verben und Adjektive induzieren ebenfalls Typen und benötigen daher eine Typdeklaration:

[5] Verbdefinition in Abschnitt 5.3, Glossardefinition in Abschnitt A.2.1; der erste Beispielsatz wurde ergänzt

Eine Inbetriebnahme ist ein Prozess.
Eine Vorschriftsmäßigkeit ist ein Merkmal.

Wir deklarieren zunächst ausschließlich das Gerundium (die substantivierte Form) als Typ. Diese benötigen wir um Verben und Adjektive selbst näher beschreiben zu können. Später definieren wir Aliase für die Substantivierungen, um Verben und Adjektive als Prädikat eines Satzes verwenden zu können.

In Fällen wie *Pumpen*, in denen der Plural eines Substantivs mit dem Gerundium eines Verbs zusammenfällt, muss darauf geachtet werden, eine Variante der Substantivierung zu wählen, für die keine Typkollisionen vorliegen. Dank der Namensanalyse fallen solche Fälle auch bei nachträglichen Ergänzungen auf. Dies ist möglich, da Substantive, Verben und Adjektive jeweils einen eigenen gemeinsamen Obertyp besitzen. Das Typsystem verfügt über einen gemeinsamer Top-Typ $\top$ aller Typen, dessen Literal der aber nicht als Obertyp deklariert werden kann.

Fest vordefinierte Typen im Typsystem sind:

- Objekt
- Prozess
- Merkmal

5.2 Untertypen und Polymorphie

Wir wollen Synonyme für Wörter benutzen können. Weiter wollen wir Bezeichner zusammen mit einem Verb benutzen können, ohne ihren exakten Typ benennen zu müssen. Dann können wir ihre Definition nachträglich und modular verändern, ohne jede Benutzung abändern zu müssen. Verben sollen nicht überladen werden müssen um eine Überbestimmung in ihren Parametertypen auszugleichen.

Wir erlauben daher die Definition von Obertypen bei der Deklaration eines Typs, also:

Eine Pumpe ist ein Objekt.

Dies deklariert den Typ *Pumpe* als Untertyp des Typs *Objekt.*

Obertypen können mehrfach deklariert werden:

Eine Pumpe ist ein Objekt.
Eine Pumpe ist ein System.

Dies deklariert den Typ *Pumpe* als Untertyp sowohl zu *Objekt* als auch *System.* Die so deklarierten Obertypen werden zu einer Menge zusammengefasst. Nach Aufsammeln aller Typdeklarationen in der Anforderungsbeschreibung

Die Syntax für Untertypsdeklarationen lautet:

```
TDecl ::= PP ('ist'|'sind') PP '.'
```

5.3 Verbdefinitionen

Verben besitzen in der Regel mehrere Parameter (Valenz) von unterschiedlichem Typ. Die Typen dieser Parameter sollen deklariert und durch unseren Übersetzer geprüft werden. Die Argumente einer Verbalphrase können zudem mit Präpositionen beschriftet sein (siehe Abschnitt 3.1). Die Anzahl der Parameter, ihr Typ und ihre Beschriftung werden in der Verbdefinition festgelegt. Die Verbdefinition konstruiert daraus einen zuvor deklarierten Typ .

Verben sowie Adjektive, deren Partizipien das Prädikat eines Satzes bilden können (z.B. *energieeffizient sein*), benötigen zudem Aliase für verschiedene Flexionsformen (Beugungen). Ein einzelnes Verb oder Adjektiv umfasst 3 Untertypen: eine für jede Zeitform (siehe Abschnitt 3.3).

Die konkrete Syntax für Typdeklarationen und Verbdefinitionen lautet:

```
VDef ::= PP ('ist'|'sind') ':' VP '.'
```

Zu beachten ist, dass der Obertyp aus Lesbarkeitsgründen bei Verbdefinitionen auf der rechten Seite des *ist* steht.

Beispiel[6]:

> *Ein Füllen ist: die Pumpe soll mit einem Fördermedium gefüllt sein.*
> *Ein Betreiben ist: der Frequenzumrichter kann als Generator betrieben werden.*

In Abschnitt 3.1 haben wir erklärt, Präpositionen als Beschriftungen für Argumente von Verben zu benutzen. Diese Beschriftungen sind spezifisch für jedes Verb. Sie können spezifisch für jede Konjugationsform definiert werden; in der Regel sind sie jedoch für alle Konjugationen jeweils in der Aktiv- und der Passiv-Form eines Verbs identisch (*betreiben* bzw. *betrieben werden*).

Bei der Definition von Verben wird die Namensanalyse außer Kraft gesetzt: die Form des Artikels wird ignoriert.

Repräsentation in der Definitionstabelle Aus den im Beispiel gezeigten Definitionen leiten wir die in Tabelle 4 dargestellte Repräsentation in der Definitionstabelle ab.

5.4 Typberechnungen

Wir definieren Typen als Mengen von Eigenschaften. Sei E die Grundmenge aller atomaren Eigenschaften (die Literale der Typnamen), und sei ein Typ $A \subseteq E$ eine Teilmenge aller möglichen Eigenschaften, dann ist die Berechnung, ob der A Untertyp $\sqsubseteq$ von $B \subseteq E$ ist, abbildbar auf die Teilmengeneigenschaft:

$$A \sqsubseteq B \Leftrightarrow closure(A) \supseteq closure(B)$$

[6] siehe Abschnitt A.2.1, Glossareinträge 7 und 16

id	VName	VType	VLabels	VArgs
#1	*gefüllt sein*	`Ausgeben`	$\{\varepsilon \Rightarrow \#2, \text{mit}\Rightarrow \#3\}$	
#2				[`Pumpe`]
#3				[`Fördermedium`]
#4	*betrieben werden*	`Betreiben`	$\{\varepsilon \Rightarrow \#5, \text{als}\Rightarrow \#6\}$	
#5				[`Freqenzumrichter`]
#6				[`Generator`]

Tabelle 4. Verben *gefüllt sein* und *betrieben werden* in der Definitionstabelle

Dazu sammeln wir zuerst alle Typdeklarationen durch den Übersetzer auf, bilden den transitiven Abschluss über die Obertypsrelation (Abschnitt 5.2), und das Ergebnis als $closure(A)$ für jeden Typ A in der Definitionstabelle. Enthielten A und B zuerst nur eine konkrete Eigenschaft, nämlich die Mitgliedschaft im Typ ihres Namens, d.h. $A = \{a\}, B = \{b\}$ mit $a, b \in E$, so ergänzt die Abbildung $closure :: E \to E$ alle diejenigen Eigenschaften, die in Obertypen von A bzw. B enthalten sind. Das Typsystem bildet einen Verband mit den Operationen Typerweiterung $\sqcup$ und kleinster gemeinsamer Obertyp $\sqcap$, $\top = \emptyset$, $\bot = E$. Der gemeinsame Obertyp $\top = \emptyset$ ist die Schnittmenge aller atomaren Eigenschaften; der gemeinsame Untertyp aller Typen $\bot = E$ ist die Vereinigung aller atomaren Eigenschaften, also die Menge aller möglichen Eigenschaften. Für domänenspezifische Anwendungen kann es sinnvoll sein, Kriterien für den gegenseitigen Ausschluss von Eigenschaften zu definieren. Da wir in dieser Arbeit keine Codegenerierung betrachten, ist es unerheblich, ob $\bot$ tatsächliche Werte repräsentieren kann.

Wird ein Verb im Text aufgefunden, wird seine Definition aus der Definitionstabelle als erwarteter (a-posteriori) Typ geladen. Der a-priori Typ ist die Liste der PP. Um zu überprüfen, ob der a-priori Typ auf den a-posteriori Typ anpassbar ist, wird die Liste der PP von links nach rechts darauf überprüft, ob sich das jeweilige Paar von Beschriftung (*label*) und Argument in der Definition des Verbs findet. Dabei wird mittels einer Hilfsfunktion die Multiplizität von Präpositionen (Beschriftungen) berechnet und daraus der a-posteriori Typ für das jeweilige Argument mit dem benutzten Parameter bestimmt.

Beispiel:

> *Ein Ausgang ist ein Anschluss. Ein Anschluss ist ein Objekt. Die Lasten sind eine Last. Eine Last ist ein Objekt. Ein Anschließen ist ein Prozess.*

Eine Untertypsbeziehung wird für diesen Ausschnitt des Glossars (siehe Abbildung 1 und Abschnitt A.2.1) wiefolgt berechnet:

$$Ausgang \sqsubseteq Anschluss$$

$$\Updownarrow$$

$$\{Ausgang, Anschluss, Objekt\} \supseteq \{Anschluss, Objekt\}$$

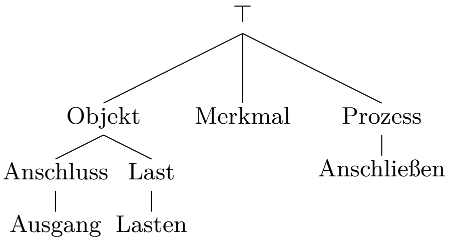


Abb. 1. Ausschnitt der Typhierarchie

Multiplizität von Argumenten Mehrfach auftretende Beschriftungen behandeln ihre Argumente positional; unterschiedliche Beschriftungen können dagegen beliebig untereinander vertauscht werden. In der Regel muss nur das unbeschriftete ε-Argument mehrfach auftreten. Seltener findet sich noch ein mehrfaches Auftreten des Präposition *von*, *durch* oder *zwischen* in einem Satz. Wir behandeln alle Präpositionen gleich, und erlauben eine beliebige Anzahl an Wiederholungen.

6 Schlussfolgerungen und Ausblick

Der Beitrag dieser Arbeit zur Unterstützung der Anforderung besteht in folgenden Punkten: zuerst definieren wir eine Grammatik für Schablonen zur Anforderungsbeschreibung. Aus der Grammatik erzeugen wir einen Parser, um syntaktische Korrektheit zu automatisch überprüfen, statt lediglich *best-practice* Regeln zu befolgen (Abschnitt 3). Wir führen eine Namensanalyse auf Bezeichnern (benannten Argumenten von Verben) durch (Abschnitt 4). Weiter führen wir eine Typprüfung auf Bezeichnern und Parametern von Verben durch (Abschnitt 5.3), um deren richtige Kombination sicherzustellen. Die Syntax für Typdeklarationen ist eine Teilmenge der natürlichen Sprache (Deutsch), genauso wie die Syntax für Terme (Anforderungen), und das obwohl Typdeklarationen ein Konzept sind, das in natürlicher Sprache nicht vorkommt. Damit erhalten wir ein Werkzeug, das das Schreiben von konsistenten Anforderungsdokumenten maßgeblich vereinfachen kann.

Diese Arbeit beschreibt Maßnahmen zur Konsistenzsicherung und -prüfung für die textuelle Form von Anforderungen, jedoch nicht für den Entwurf, den die Anforderungen beschreiben. Die kontrollierte Sprache sollte daher eingebettet werden in ein System zur Analyse der Zusammenhänge zwischen Anforderungen. Da die Zusammenhänge zwischen Anforderungen ebenfalls in natürlicher Sprache geschrieben werden (Bedingte Anforderungen, Modalverben) können die dafür notwendigen Informationen entweder aus der abstrakten Syntax entnommen oder mittels Codegenerierung für eine Zwischensprache zur Anforderungsmodellierung in ein Zwischenformat ausgeben werden.

Codegenerierung soll die Anbindung der Anwendungsspezifikation an andere Artefakte des Entwicklungsprozesses (z.B. VHDL-Code in technischen Systemen) ermöglichen, die einen höheren Grad an Verfeinerung aufweisen, als sich dies sinnvoll in natürlicher Sprache beschreiben lässt. Dazu stellt man idealerweise eine bidirektionale Korrespondenz mittels Parser und Codegenerator für die Zielsprache her, so dass Änderungen (am Entwurf bzw. der Implementierung) in beide Richtungen propagiert werden können.

Für die weitere Entwicklung des gesamten Ansatzes, natürliche Sprache für domänenspezifische Modellierung einzusetzen, ist es interessant, einen Blick auf andere Domänen zu werfen als die in dieser Arbeit fokussierten technischen Systeme. Interessante Domänen sind z.B. die Softwareentwicklung (starke Ähnlichkeit zu technischen Systemen), Kochrezepte (schablonenbasierte Syntax, nichttechnischer Natur) oder juristische Texte (fokussiert auf Syntax und Äquivalenz auf Termebene anstatt abstrakter Assoziation).

Das vorgestellte Typsystem basiert auf expliziten Deklarationen. Hier wäre stattdessen der Einsatz von Typinferenz möglich.

Danksagung Diese Arbeit entstand im Rahmen des vom Bundesministerium für Bildung- und Forschung (BMBF) geförderten Forschungsprojekts ELSY (Nr. 16M3202D).

Literaturverzeichnis

[Ambriola u. Gervasi 2006] Ambriola, Vincenzo ; Gervasi, Vincenzo: On the Systematic Analysis of Natural Language Requirements with Circe. In: *Automated Software Engineering* 13 (2006), S. 107–167

[Cowan 2000] Cowan, John W.: *The Complete Lojban language*. The Logical Language Group, Inc., 2000. – ISBN 0–9660283–0–9

[Farfeleder 2012] Farfeleder, Stefan: *Requirements Specification and Analysis for Embedded Systems*, Technische Universität Wien, Diss., November 2012

[Hull u. a. 2005] Hull, Elizabeth ; Jackson, Ken ; Dick, Jeremy: *Requirements Engineering*. Springer, 2005. – ISBN 978–1–85523–879–4

[KSB 2014] KSB: *Betriebs-/ Montageanleitung PumpDrive 2*. KSB Aktiengesellschaft, Johann-Klein-Str. 9, 67225 Frankenthal, Juni 2014. http://www.ksb.com/propertyblob/2286696/data/PumpDrive%202.pdf

[Kuhn u. Bergel 2014] Kuhn, Tobias ; Bergel, Alexandre: Verifiable source code documentation in controlled natural language. In: *Sciene of Computer Programming* 16 (2014), S. 121–140

[Lachowicz u. a. 2015] Lachowicz, Dom ; Figuière, Hubert ; Sevior, Martin: *Link Grammar*. http://www.abisource.com/papers/. Version: August 2015

[Palmer u. a. 2010] Palmer, Martha ; Gildea, Daniel ; Xue, Nianwen: Semantic Role Labeling. In: *Synthesis Lectures on Human Lanugage Technologies* (2010)

[Rupp 2014] Rupp, Chris: *Requirements-Engineering und -Management*. Hanser, 2014. – ISBN 978–3–446–43893–4

A Praxisbeispiel

Wir zeigen am Beispiel der *Betriebs-/ Montageanleitung* des *PumpDrive2* [KSB, 2014], einem Frequenzumrichter zur Steuerung von Pumpenaggregaten, wie die kontrollierte Sprache zur Modellierung von Anforderungen eingesetzt wird.

In Abschnitt A.1 ist der Originalausschnitt wiedergegeben. Danach folgt in Abschnitt A.2 der in kontrollierte Syntax umformulierte Text. Abschnitt A.2.1 enthält das daraus erarbeitete Glossar. In Abbildung 2 ist die deklarierte Typhierarchie abgebildet.

A.1 Original-Abschnitt: 7 Inbetriebnahme / Außerbetriebnahme[7]

Vor Inbetriebnahme müssen folgende Punkte sichergestellt sein:

- Pumpe ist entlüftet und mit Fördermedium gefüllt.

[7] vgl. [KSB, 2014] S. 48

- Pumpe wird nur in Auslegefliessrichtung durchströmt, um einen generatorischen Betrieb des Frequenzumrichters zu vermeiden.
- Ein plötzliches Anfahren des Motors bzw. des Pumpenaggregats verursacht keine Schäden an Personen und Maschinen.
- Es sind keine kapazitiven Lasten z.B. zur Blindstromkompensation an den Ausgängen des Geräts angeschlossen.
- Die Netzspannung entspricht dem für den Frequenzumrichter zugelassenen Bereich.
- Der Frequenzumrichter ist vorschriftsmäßig elektrisch angeschlossen (⇒ Kapitel 5.4 Seite 21)
- Freigaben und Startbefehle, die den Frequenzumrichter starten können, deaktiviert sind (siehe Digitaleingänge DI-EN Digitaler Freigabe-Eingang und DI1 Anlagenstart).
- Am Leistungsmodul des Frequenzumrichters liegt keine Spannung an.
- Der Frequenzumrichter bzw. das Pumpenaggregat darf nicht über die zugelassene Nennleistung belastet werden.

A.2 Abschnitt 7 in kontrollierter Syntax

Wenn eine Pumpe in Betrieb genommen werden soll, dann:

- die Pumpe muss entlüftet sein
- die Pumpe muss mit einem Fördermedium gefüllt sein
- die Pumpe darf nicht entgegen Auslegefliessrichtung durchströmt werden
- der Frequenzumrichter darf nicht als Generator betrieben werden.

Wenn die Pumpe in Betrieb genommen werden soll, und wenn ein Motor angefahren wird, dann:

- keine Schäden dürfen an Personen verursacht werden
- keine Schäden dürfen an Maschinen verursacht werden.

Wenn die Pumpe in Betrieb genommen werden soll, dann dürfen keine Lasten, die kapazitiv sind, an den Ausgängen, die durch den Frequenzumrichter besessen werden, angeschlossen sein. Eine Blindstromkompensation soll eine Last, die kapazitiv sein soll, beschreiben.

Die Netzspannung muss dem Bereich, für den der Frequenzumrichter zugelassen ist, entsprechen.

Der Frequenzumrichter muss angeschlossen werden. Das Anschließen, das die Elektrizität betrifft, muss vorschriftsmäßig sein.

Wenn die Pumpe in Betrieb genommen werden soll, dann:

- die Freigaben, die den Frequenzumrichter starten können, müssen deaktiviert sein
- die Startbefehle, die den Frequenzumrichter starten können, müssen deaktiviert sein
- am Leistungsmodul, das durch den Frequenzumrichter besessen wird, soll keine Spannung anliegen.

Der Frequenzumrichter darf nicht oberhalb der Nennleistung, für die der Frequenzumrichter zugelassen ist, belastet werden.

Das Pumpenaggregat darf nicht oberhalb der Nennleistung, für die das Pumpenaggregat zugelassen ist, belastet werden.

A.2.1 Notwendige Glossareinträge für Abschnitt 7

Hinweis: die Typen *Objekt*, *Merkmal* und *Prozess* sind vordefiniert, siehe Abschnitt 5.1.

Eine Pumpe ist ein System.

Eine Inbetriebnahme ist: die Pumpe soll in Betrieb genommen werden[8]. Eine Inbetriebnahme ist ein Prozess. Ein Betrieb ist eine Inbetriebnahme.

Ein Entlüften ist: die Pumpe soll entlüftet werden. Ein Entlüften ist ein Prozess.

Ein Füllen ist: die Pumpe soll mit einem Fördermedium gefüllt sein. Ein Füllen ist ein Prozess. Ein Fördermedium ist ein Material. Ein Material ist ein Objekt.

Ein Durchströmen ist: die Pumpe kann entgegen der Auslegefliessrichtung durchströmt werden. Ein Durchströmen ist ein Prozess. Eine Auslegefliessrichtung ist eine Richtung. Die Richtung ist eine Regel. Eine Regel ist ein Objekt.

Ein Betreiben ist: der Frequenzumrichter kann als Generator betrieben werden. Ein Betreiben ist ein Prozess. Ein Frequenzumrichter ist ein System. Ein Generator ist ein System. Ein System ist ein Objekt.

Ein Anfahren ist: der Motor soll angefahren werden. Ein Anfahren ist ein Prozess. Ein Motor ist ein System.

Ein Verursachen ist: Die Schäden an einem Objekt können verursacht werden. Ein Verursachen ist ein Prozess. Die Schäden sind ein Objekt. Die Personen sind ein Objekt. Die Maschinen sind ein Objekt.

Eine Kapazitivität ist: die Lasten können kapazitiv sein. Eine Kapazititivät ist ein Merkmal. Die Lasten sind eine Last. Eine Last ist ein Objekt.

Ein Besitzen ist: durch den Frequenzumrichter können die Ausgängen besessen werden. Ein Besitzen ist ein Prozess. Die Ausgängen sind ein Ausgang. Ein Ausgang ist ein Anschluss. Ein Anschluss ist ein Objekt.

Ein Anschließen ist: die Lasten können an den Ausgängen angeschlossen sein. Ein Anschließen ist ein Prozess.

Ein Beschreiben ist: eine Blindstromkompensation kann eine Last beschreiben. Ein Beschreiben ist ein Prozess. Eine Blindstromkompensation ist ein Prozess.

Ein Entsprechen ist: die Netzspannung muss einem Bereich entsprechen. Ein Entsprechen ist ein Prozess. Die Netzspannung ist eine Spannung. Die Spannung ist ein Objekt. Ein Bereich ist eine Regel.

[8] Das Verb lautet *genommen*; wir bilden es auf das Prädikat *Inbetriebnahme* ab

Ein Zulassen ist: für einen Bereich ist ein Frequenzumrichter zugelassen. Ein Zulassen ist ein Prozess.

Ein Anschließen ist: der Frequenzumrichter kann angeschlossen sein. Ein Anschließen ist ein Prozess.

Eine Vorschriftsmäßigkeit ist: das Anschließen muss vorschriftsmäßig sein. Eine Vorschriftsmäßigkeit ist ein Merkmal.

Ein Anschließen ist: die Elektrizität kann angeschlossen sein. Die Elektrizität ist ein Objekt.[9]

Ein Starten ist: die Freigaben können den Frequenzumrichter starten. Ein Starten ist ein Prozess. Die Freigaben sind eine Freigabe. Eine Freigabe ist eine Regel.

Ein Starten ist: die Startbefehle können den Frequenzumrichter starten. Die Startbefehle sind ein Startbefehl. Ein Startbefehl ist ein Ereignis. Ein Ereignis ist ein Objekt.

Ein Deaktivieren ist: die Freigaben können deaktiviert sein. Ein Deaktivieren ist ein Prozess.

Ein Deaktivieren ist: die Startbefehle können deaktiviert sein.

Ein Anliegen ist: am Leistungsmodul soll eine Spannung anliegen. Das Anliegen ist ein Prozess. Ein Leistungsmodul ist ein Objekt. Eine Spannung ist ein Objekt.

Ein Besitzen ist: ein Leistungsmodul kann durch den Frequenzumrichter besessen wird.

Ein Belasten ist: der Frequenzumrichter kann oberhalb einer Grenzleistung belastet werden. Ein Belasten ist ein Prozess. Eine Nennleistung ist eine Leistung. Eine Leistung ist eine Grenzleistung. Eine Grenzleistung ist eine Grenze. Eine Grenze ist ein Wert. Ein Wert ist ein Objekt.

Ein Belasten ist: das Pumpenaggregat kann oberhalb einer Grenzleistung belastet werden.

[9] betrifft/betreffen ist ein Platzhalter-Verb, das durch sein erstes Argument, das ein Prozess sein muss, ersetzt wird

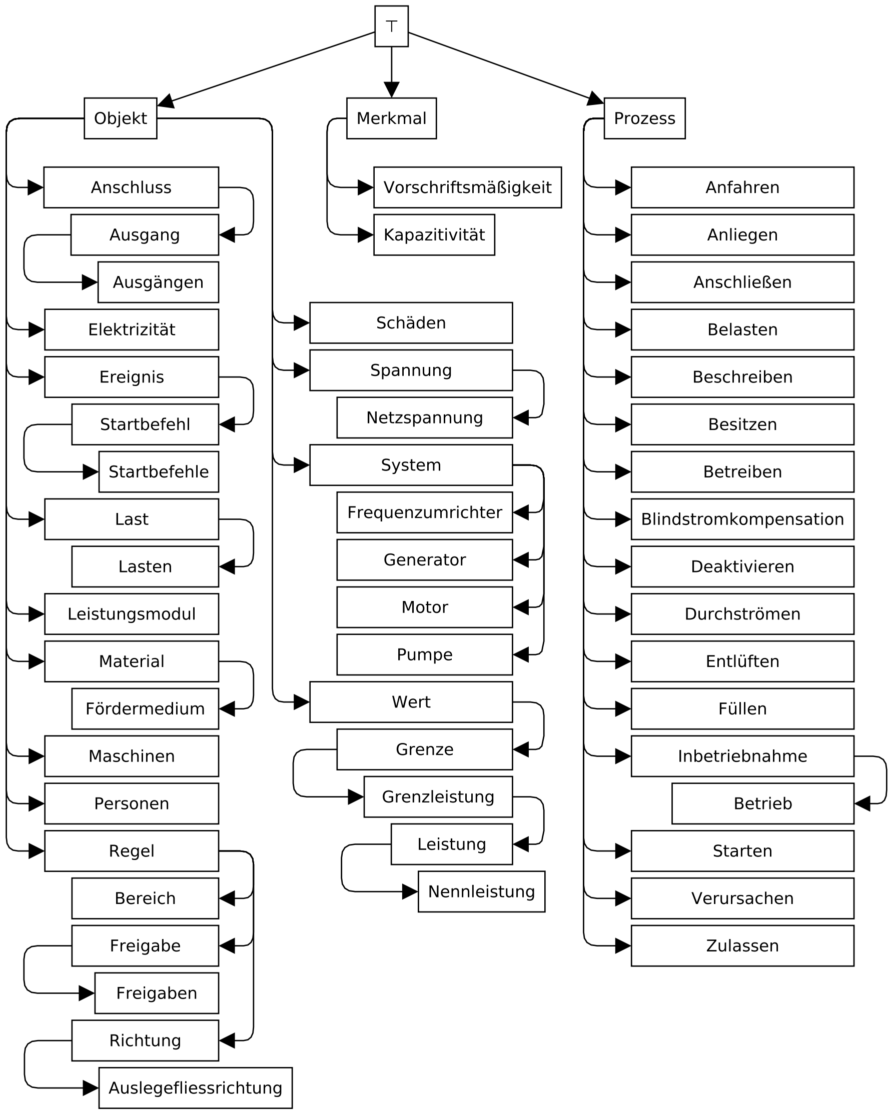


Abb. 2. Typhierarchie für Abschnitt 7

Analyse Paralleler Programme durch Kombination Automatenbasierter Erreichbarkeitsanalyse mit Datenflussanalyse
Extended Abstract

Alexander Wenner

Institut für Informatik, Fachbereich Mathematik und Informatik
Westfälische Wilhelms-Universität Münster
alexander.wenner@uni-muenster.de

Die Entwicklung paralleler Programme hat in den vergangenen Jahren stark zugenommen. Dazu haben insbesondere das Aufkommen von Programmiersprachen die Parallelität direkt unterstützen und die zunehmenden Verfügbarkeit paralleler Hardware beigetragen. Parallele Programmierung ist jedoch komplex und daher fehleranfällig. Statische Analysen können den Programmierer zur Entwicklungszeit auf mögliche Fehlerquellen aufmerksam machen und ihn damit bei der Entwicklung unterstützen.

In den letzten Jahren wurden Erreichbarkeitsprobleme für Varianten dynamischer Pushdown Netzwerke (DPN) [1], als abstraktes Modell für den Kontrollfluss paralleler rekursiver Programme, untersucht. DPN sind eine Erweiterung von Kellerautomaten, die mehrere parallel arbeitende Stacks modellieren, deren Anzahl sich zu Laufzeit dynamisch ändern kann. Im Gegensatz zu der Unentscheidbarkeit von Erreichbarkeit für starke Synchronisationsprimitive [2] bleibt Entscheidbarkeit erhalten wenn Synchronisation durch well-nested Locking [3] realisiert wird. Dies kann um zusätzliche einfache Joins erweitert werden [4]. Alternativ ist Erreichbarkeit auch für contextual Locking und einfache Joins entscheidbar [5].

Wir erweitern zunächst das Modell aus [4] um ausdrucksstärkere Joins und zeigen, dass Erreichbarkeit präzise entscheidbar bleibt. Hierzu adaptieren wir die automatenbasierten Techniken aus [4] und zeigen, dass die Menge kritischer Ausführungen durch eine reguläre Menge von Bäumen dargestellt werden können. Erreichbarkeit kann damit durch den Leerheitstest eines Baumautomaten überprüft werden.

In einem zweiten Schritt präsentieren wir eine Möglichkeit, unter Verwendung des obigen Resultats, Datenflussanalysen für sequentielle Programme auf parallele Programme zu erweitern. Verschiedene existierende Ansätze [6,7,8] basieren auf der Idee einzelne Prozesse in einem parallelen Programm zu betrachten und interferierende Operationen paralleler Prozesse als zusätzliche Operationen in die Datenflussanalyse einfließen zu lassen. Aufbauend auf dieser Idee entwickeln wir ein Framework für Datenflussanalysen in dem wir die Erreichbarkeitsanalyse für DPN verwenden um die Interferenz zwischen Prozessen in parallelen Programmen zu approximieren. Zu diesem Zweck konstruieren wir eine Folge von Abstraktionen, im Kontext von Abstrakter Interpretation [9], die es uns erlauben einzelne sequentielle Prozesse zu betrachten und gleichzeitig ein DPN zu extrahieren mit dem wir Interferenzen extrahieren können.

In Kombination erhalten wir eine Vorgehensweise um Datenflussanalysen für sequentielle Programme auf parallele Programme zu erweitern.

Literatur

1. Bouajjani, A., Müller-Olm, M., Touili, T.: Regular symbolic analysis of dynamic networks of pushdown systems. In: CONCUR 2005. LNCS 3653, Springer (2005)
2. Ramalingam, G.: Context-sensitive synchronization-sensitive analysis is undecidable. ACM Trans. Program. Lang. Syst. **22**(2) (2000)
3. Lammich, P., Müller-Olm, M., Wenner, A.: Predecessor sets of dynamic pushdown networks with tree-regular constraints. In: CAV 2009. Volume 5643 of Lecture Notes in Computer Science., Springer (2009) 525–539
4. Gawlitza, T.M., Lammich, P., Müller-Olm, M., Seidl, H., Wenner, A.: Join-lock-sensitive forward reachability analysis for concurrent programs with dynamic process creation. In: VMCAI 2011. Volume 6538 of Lecture Notes in Computer Science., Springer (2011) 199–213
5. Lammich, P., Müller-Olm, M., Seidl, H., Wenner, A.: Contextual locking for dynamic pushdown networks. In Logozzo, F., Fähndrich, M., eds.: Static Analysis. Volume 7935 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2013) 477–498
6. Miné, A.: Static analysis of run-time errors in embedded critical parallel C programs. In: Proc. of the 20th European Symposium on Programming (ESOP'11). Volume 6602 of Lecture Notes in Computer Science (LNCS)., Springer (Mar. 2011) 398–418 http://www.di.ens.fr/ mine/publi/article-mine-esop11.pdf.
7. Jeannet, B.: Relational interprocedural verification of concurrent programs. Software and System Modeling **12**(2) (2013) 285–306
8. Seidl, H., Vene, V., Müller-Olm, M.: Global invariants for analyzing multithreaded applications. Proc. of the Estonian Academy of Sciences: Phys., Math. **52**(4) (2003)
9. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, ACM (1977)

dsOli2: Discovery and Comprehension of Interconnected Lists in C Programs

David H. White, Thomas Rupprecht and Gerald Lüttgen

Software Technologies Research Group, University of Bamberg, 96045 Germany

Abstract. Comprehension of C programs can be a challenging task, especially when they contain pointer-based dynamic data structures. Based on prior experience with our tool dsOli1, we report on work in progress concerning a new dynamic analysis for automated data structure identification that targets C source code. Our technique first applies a novel abstraction on the evolving memory structures observed at runtime for discovering the building blocks of complex data structures. By analyzing the interconnections between the building blocks, we are then able to identify trees, doubly-linked lists, skip lists, as well as relationships between these such as nesting. We give preliminary results from a prototype implementation, which aims to provide a natural language description of the identified data structures. This information will benefit software developers when code must be comprehended or modified.

1 Introduction

C programs are notoriously difficult to comprehend, and this is especially true for legacy or low-level code, e.g., that found in OSs or device drivers. In such situations it is not uncommon to see programmers employ complex usages of pointers, types and memory allocation to achieve the desired behavior or efficiency. These constructs are often used to implement the dynamic data structures of a program, and thus data structures can form a major obstacle in program comprehension, optimization and verification. To partially alleviate this obstacle we propose a dynamic analysis for automatic identification of dynamic data structures in C programs.

The essence of our analysis is to first discover the building blocks of complex data structures, which are essentially singly linked lists (SLLs), and then to analyze any relationships that exist between the lists. Lists may be either *tightly connected*, where they comprise some part of a more complex data structure, e.g., the two lists running in reverse directions through a doubly-linked list (DLL), or *loosely connected*, where they describe relationships between specific data structures, e.g., the parent-child relationship found in nested lists.

The identification of dynamic data structures is made challenging due to manipulation operations that temporally transform a *stable shape* into a *degenerate shape*. For example, consider how the key feature of a DLL is broken during the insertion of a node; if one were to inspect the shape at such an intermediate state, then it may be difficult to give the correct *label*, i.e., name of the data structure. Approaches such as dsOli1 [11] and DDT [7] handle this by trying to find data structure operation boundaries, while MemPick [5] attempts to perform identification only in the quiescent periods of a data structure. In both cases, identification is performed when one can be reasonably sure the data structure has a stable shape.

In our work we include degenerate shapes but override their influence by observing the *context* in which a shape appears. Context arises from two sources: *structural repetition*, which occurs when there exist many structures performing the same role, e.g., the multiple child lists found in parent-child nested lists, and *temporal repetition*, which occurs when the same structures exist over multiple program time steps. By *discovering evidence* for specific occurrences of data structures and then *reinforcing* this evidence through structural and temporal repetition, our approach enables identification even when temporary degenerate shapes are encountered. To illustrate the utility of our approach, we track variables that represent entry points to dynamic data structures and aim to annotate these with natural language descriptions of the reachable data structure, e.g., "Entry point `p` points to a skip list with a parent child nesting to DLLs".

The remainder of this paper is organized as follows. In Sec. 2 we discuss the complexities of data structures in C heaps, which motivates many of the design decisions we have made for dsOli2. Sec. 3 describes our approach from a high level with an illustrative example, and in Sec. 4 we dive into the details. We report preliminary results in Sec. 5 obtained from our prototype implementation, and finally present conclusions and future work in Sec. 6.

Related Work. Our dynamic analysis aims to identify data structures but provides no soundness guarantee. In contrast, modern shape analysis tools, such as Predator [4] and Forester [6], are sound and employ symbolic execution to learn shape predicates that allow memory safety to be checked automatically. In particular, Forester summarizes repetitive graph structures with forest automata to handle skip lists and trees. However, neither approach can handle the recursion commonly found in tree operations, and as their focus is memory safety, it is not clear how naturally the learnt shape predicates fit the goal of program comprehension.

The dynamic analyses HeapDbg [8] and ARTISTE [3] represent multiple concrete data structure nodes with a single abstract node, which is in turn checked for interesting shape properties. Our approach shares much in common with the techniques of HeapDbg: their summarization process employs structural and temporal repetition, but as the process is conservative, temporal joins force the label of an abstract node to be reduced to the most general available. While this works for HeapDbg's tree label, Artiste includes DLLs and, if temporal joins were to be performed, then the precision of the DLL label would be lost.

MemPick [5] functions on object code and excels in distinguishing different types of trees. In contrast, we require source code but handle skip lists and produce a much richer description of the connections between data structures. Finally, dsOli1 [11] and DDT [7] go beyond all these approaches in that they

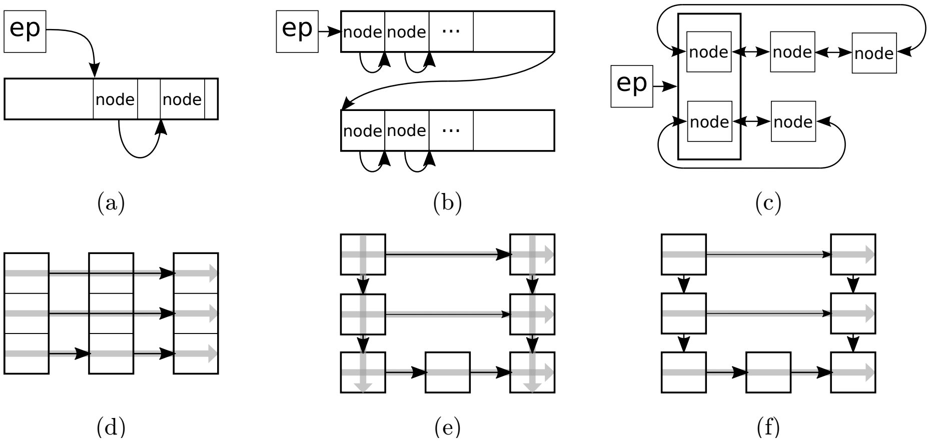


Fig. 1. Complexities of C heaps: (a) custom allocator, (b) cache efficient list [2], (c) Linux kernel DLL [1], (d-f) skip lists where building block lists are indicated by bold gray arrows. Examples of (d) and (e) appear in `tests/skip-list/jonathan-skip-list.c` and `tests/forester-regre/test-f0021.c` of Predator [4], respectively.

also seek to discover the operations that manipulate the data structures. DDT accomplishes this by assuming that data structures are accessed via well-defined interface functions, while dsOli1 employs a machine learning approach to locate repetitive code segments indicative of operations. Compared to dsOli1, we currently don't consider operations but do expand the variety of data structures that are in-scope considerably.

2 Heap Usage in C Programs

The type safety of modern programming languages such as Java and C# constrains the actions that a programmer may take and results in programs having relatively well structured heaps. However, in languages frequently used for OS programming such as C, where pointer arithmetic and type casting may be freely applied and memory management is in the hands of the programmer, the heap can be formed in a more ad-hoc manner. In this section we describe some of the challenging C code we have seen in practice that leads us to this conclusion, and in the next section we outline how our approach copes with this challenge. Firstly, we briefly introduce the notion of a *points-to graph*, which describes a snapshot of program memory by representing *memory chunks*, i.e., stack/global variables and dynamically allocated memory, as vertices and *pointers* as edges.

A typical assumption is that a memory chunk represents a single node of a data structure; however, in practice this is broken in a number of situations. Firstly, if a custom memory allocator is employed, but memory chunks are detected at the level of the system memory allocator, then it may be the case that multiple nodes of potentially multiple data structures appear in the same memory chunk (Fig. 1(a)). Secondly, cache-efficient data structures combine multiple nodes into a single memory chunk to enhance performance (Fig. 1(b)). Thirdly, head nodes of multiple lists may be embedded in the same memory chunk (Fig. 1(c)). This is common practice with the cyclic DLL type `struct list_head` employed by the Linux kernel [1], which is designed to be embedded inside another struct. Given this *cyclic* property, a natural interpretation is to treat the head node uniformly with the remainder of the list. This gives rise to an alternative view, i.e., as a list where the nodes occupy memory chunks of varying sizes. In the above case, a list of length n consists of one node in a memory chunk of type t_1 and $n-1$ nodes each in a memory chunk of type t_2. Macros are provided that allow the outer struct to be reached from a list head struct via pointer arithmetic and casting.

The key insight to model all of the above situations uniformly is to relax the assumption that a list linkage offset should occur at a fixed offset from the memory chunk start address. Thus, it is necessary to track lists in terms of their linkage rather than in terms of memory chunk type. In the next section we show how our approach handles this by determining the minimal subregions of memory chunks needed to establish list linkage.

Now that we have discussed the complexities surrounding list formation, we turn to how lists are connected. A connection may be made either by *overlay*, where at least one node from each list occupies the same memory chunk, or by *indirection*, where there exists a pointer, or a chain of pointers, from the memory chunk holding the node of one list to a memory chunk holding a node of another list. To illustrate this we consider possible skip list constructions. Firstly, if the number of levels are known *a priori*, then it is common to employ a memory chunk with an array of linkages, where the array element at index i represents the linkage to the next node at level i (Fig. 1(d)). Thus, in situations where multiple levels run through the same node, these are connected by overlay. Secondly, all nodes in the skip list may be of the same type; in other words, each memory chunk has a `next` pointer to the next node of the level it represents and a `down` pointer to the level below (Fig. 1(e)). Since all nodes are of the same type, atomic lists are formed both in the horizontal and vertical directions and are again connected by overlays. Lastly, consider a skip list where each level is represented by a node of different type (Fig. 1(f)). Since only the horizontal linkage forms lists, the downward link is an indirect connection between lists.

In the next section we show how our approach uniformly handles the variety of implementation techniques that may be employed by firstly gathering evidence and then employing structural and temporal repetition to consolidate the acquired evidence.

3 Overview of our Approach

In this section we give an overview of our approach and provide motivation with the simple example in Fig. 2, which also shows our approach as a pipeline.

Table 1. Memory structures (with abbreviations) in-scope for our approach, plus the number of strands required for discovery and the priority in the identification phase. Memory structures typically have several implementations, in the case of Head/Tail Pointers this affects the category. Sharing denotes two lists which share a downstream cell sequence, while intersecting lists is a catch-all for any pair of connected strands.

Data	**# Strands**	**Priority**	**Connection**	**# Strands**	**Priority**
Structure	**Required**			**Required**	
(Cyclic) SLL	1	-	Head/Tail Ptrs. (HT)	1 or 2+	3
(Cyclic) DLL	2	1	Parent Pointers (PP)	2+	4
Tree	2+	2	Intersecting Lists (IL)	2	7
Skip List (SL)	2+	5	Nesting (N)	2	8
Grid	2+	6	Sharing	2	9

The example shows two time steps in the construction of a SLL of DLLs; note that at time step t, there exists a degenerate DLL child. In favor of a succinct explanation, details are delayed until Sec. 4.

We commence from the classic definition of an SLL, which is a sequence of memory chunks all of the same type, where the entirety of each chunk constitutes one node in the list. A subset of pointers between these chunks fulfill a *linkage condition*, which states that all pointers originate at the same *linkage offset* from the start of the chunk and terminate at the start address of the next chunk.

Strands. To handle the scenarios outlined in Sec. 2, we relax the notion that the nodes of the list occupy the whole memory chunk, and instead try to discover what we term *strands*, which will form the basic building blocks of the structures we seek to identify. A strand represents a sequence of *subregions of memory chunks*, each termed a *cell*, such that the same linkage condition can be established between the cells. Thus, the linkage offset is now given relative to the start address of a cell. Strands (S_i) are indicated by bold arrows in Fig. 2(a).

Strand Connections. Our approach is driven by relationships between strands, which we term *strand connections*. Each strand connection describes exactly *one* way in which the cells of two strands are related, hence multiple strand connections between a pair of strands are possible. Merging strand connections that describe the same relationship will be of key importance in the accumulation of evidence, and we define strand connections with offsets relative to the cells in order to handle the scenarios of Sec. 2. We construct a *strand graph* where vertices represent strands and edges represent strand connections, see Fig. 2(b). Since only two time steps of the program are considered in the illustrative example, it is unsurprising that both strand graphs have the same structure. For now note that strand connections with the same edge style denote the same relationship type; for example, the DLL strands form a bi-directional overlay connection, while two kinds of uni-directional indirect connections are formed between the parent SLL and each child DLL.

Memory Structures. We use the term *memory structure* to speak collectively about data structures and connections between data structures, i.e., both the tight and loose connections mentioned in Sec. 1. The list of memory

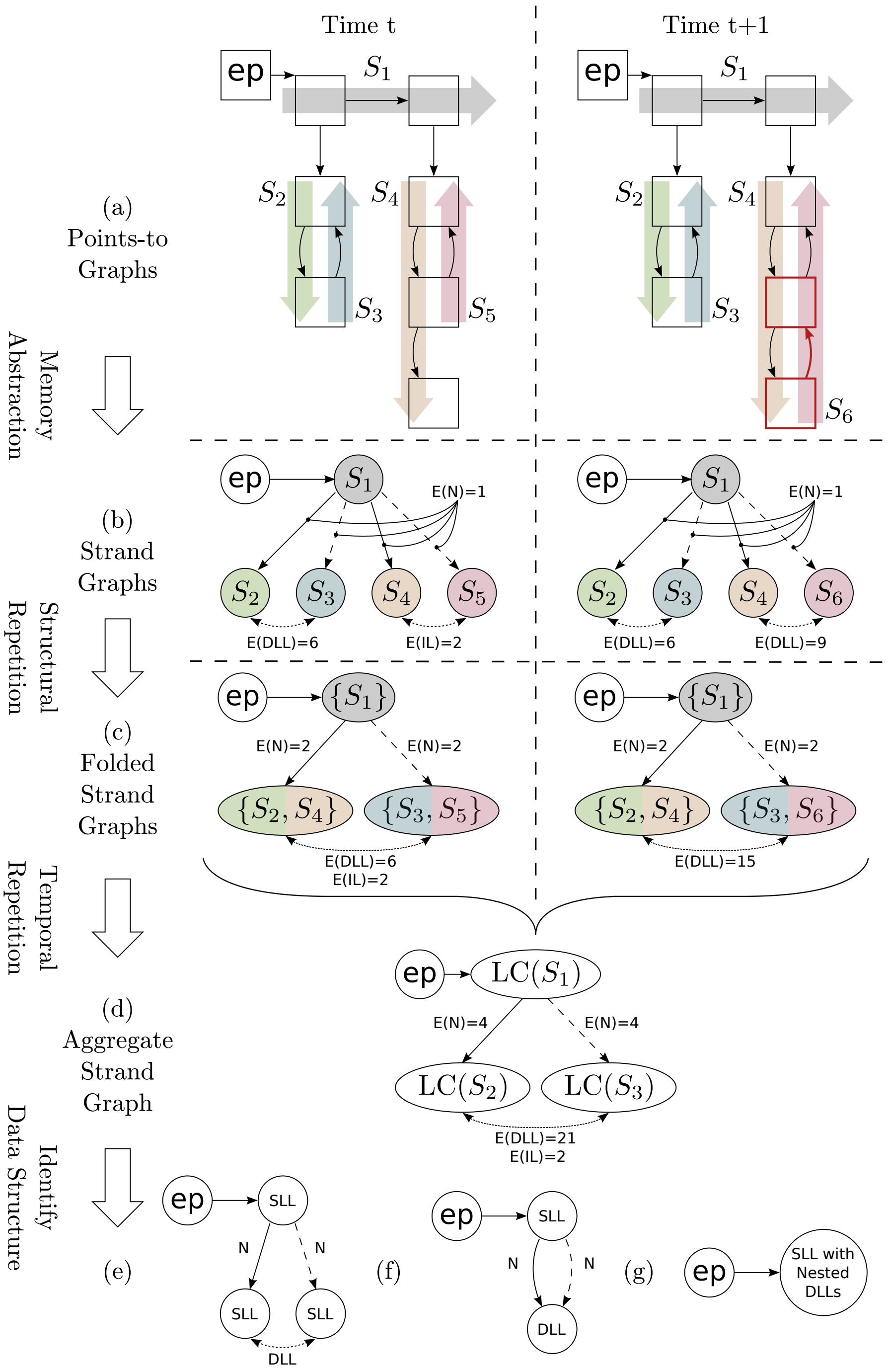


Fig. 2. Left: the pipeline of our approach, right: the illustrative example of Sec. 3.

structures in-scope for our approach are given in Table 1 and are categorized based on the number of strands required for their discovery. For example, a DLL requires exactly two strands to be discovered, while a skip list requires two or more strands. We now refer to memory structures requiring one or two strands as Category 1/2 and those requiring more than two as Category 2+. We will see later that memory structures in the Category 2+ are discovered at a later stage of the approach than those in Category 1/2.

Evidence Gathering. When a strand connection is found we immediately determine the supporting evidence for that strand connection for each Category 1/2 memory structure. The evidence is weighted by the number of cells and connections between cells that must be present for that memory structure to be correctly identified. Essentially, our goal is to count the number of things that have gone "right" for such a memory structure to exist and use this for evidence. For example, the weight of evidence gathered for nested lists on overlays, where the strands must only intersect in one memory chunk, is much weaker than that for DLLs, where the strands must form a very specific connection. Non-zero evidence is shown on the strand connections of Fig. 2(b). The degenerate DLL in the first time step has an evidence count of 2 for Intersecting Lists (IL); in this case, evidence is simply the number of overlay connections between the cells of each strand. When the DLL regains the correct shape at time $t + 1$, it has an evidence count of 9 based on the length of both composite strands $(3 + 3)$ and the number of intersection points (3). Strand connections describing nesting (N) have an evidence count of 1 as the connection is made by a single pointer.

Structural Repetition. The primary use of structural repetition is to group elements of the strand graph that perform the same role within one program time step. This grouping is realized via a merge algorithm that results in a *folded strand graph* and, since this contains merged strand connections, it serves to reinforce the evidence of Category 1/2 memory structures. Observe in Fig. 2(c) that the vertices have now become *sets* of strands. Merging partially addresses the problem of degenerate shapes, i.e., if strands with the correct shape can be grouped with those having degenerate shapes, then the majority can override the minority. The correct shape is generally in the majority since degenerate shapes are produced by manipulations that typically only have a local effect. In Fig. 2(c), this is seen between strands $\{S_2, S_4\}$ and $\{S_3, S_5\}$ at time t.

With the folded strand graph to hand, the identification of Category 2+ memory structures begins. For any suitable subgraph in the folded strand graph, it is checked if that subgraph has the property required of the corresponding Category 2+ memory structure. If found to be true, all strand connections that comprise that memory structure record the associated evidence.

Temporal Repetition. To track the temporal behavior of a memory structure and enable the identification of temporal repetition, we must determine which strands represent the same atomic component of a data structure over multiple time steps. This is a very difficult task to do globally as lists will be split, joined, created and deleted at runtime, and any labeling system will end up with some amount of discontinuity. Instead, we tackle this problem by considering the labeling from the point of view of each entry point separately, since entry points are inherently stable over their lifetimes. For each time step that an entry point exists, we extract the subgraph of the folded strand graph reachable from that entry point. The subgraphs are then merged into an *aggregate strand graph*, and thus temporal repetition is identified whenever multiple graph elements are merged together. Naturally, the evidence embedded in those elements is also merged and, hence, evidence for both Category 1/2 and Category 2+ memory structures is reinforced, further reducing the effect of degenerate shapes. Vertices of this graph become abstract descriptions of the original strands in terms of their *linkage conditions* (LC). The aggregate strand graph is shown in Fig. 2(d); note that the evidence for the DLL shape is overwhelming.

Identifying Memory Structures. The final barrier for memory structure identification arises from the fact that there may be several possible interpretations of the aggregate strand graph. We resolve this as follows: we first set the label of each strand connection in the aggregate strand graph to the one with the most evidence and set the label of all vertices to be `SLL` (Fig. 2(e)). We then group graph elements according to the priorities given in Table 1 and assign a textual label to the group. For example, DLLs have priority 1, so strand connections labeled DLL and their associated strands are grouped first (Fig. 2(f)). These elements then form an atomic vertex in subsequent groupings. Ultimately, we end up with a graph (Fig. 2(g)) of one atomic vertex with a textual label describing the whole data structure reachable from the entry point.

4 Details of our approach

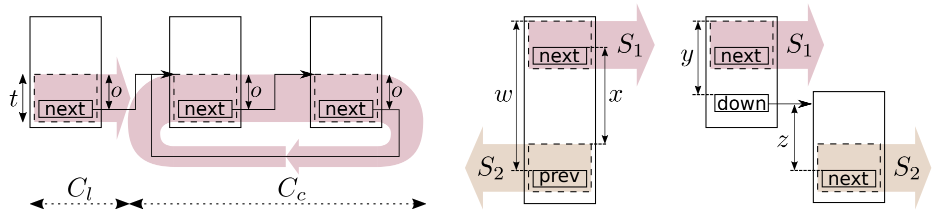
We now formalize the concepts presented in the illustrative example of Sec. 3.

4.1 Memory Abstraction

To identify the data structures employed by the program we reconstruct a sequence of points-to graphs $\langle G_0^{pt}, \ldots, G_n^{pt} \rangle$ from an execution of the program under analysis. This reconstruction is enabled by first instrumenting the program, which results in the runtime capture of *program events* such as pointer writes and dynamic memory (de)allocation. The result of the program event at *time step* t is captured by G_t^{pt}, where $1 \leq t \leq n$ and G_0^{pt} is empty.

Definition 1. *A* **points-to graph** $G^{pt} = (\mathcal{V}, \mathcal{E})$ *is a directed graph comprising a vertex set* $\mathcal{V}$ *representing memory chunks and an edge set* $\mathcal{E} \subseteq \mathcal{V} \times \mathbb{N} \times \mathcal{V} \times \mathbb{N}$ *representing pointers.*

An edge $(v_s, a_s, v_t, a_t) \in \mathcal{E}$ captures the points-to relationship between two memory chunks established by a pointer with source address a_s, encapsulated by vertex v_s, and target address a_t, encapsulated by vertex v_t. A memory chunk is either a *heap chunk* (a memory region returned from dynamic memory allocation, e.g., `malloc`) or a *stack/global chunk*. Our points-to graphs only consist of reachable memory, so if a leak occurs, then all unreachable chunks are removed.



(a) (b) (c)

Fig. 3. Details of (a) a strand $S = ((t, o), (C_l, C_c))$, (b) an overlay strand connection $S_1 \overset{wx}{\longleftrightarrow} S_2$ and (c) an indirect strand connection $S_1 \xrightarrow{yz} S_2$. Memory chunks have black outline, cells are dashed, and strands are indicated with large transparent arrows.

All memory chunks are typed with standard C types, and a heap chunk becomes typed when it is accessed by a non void * pointer. Usages of a memory chunk must be typed consistently, i.e., if a memory address a is accessed via pointer types `t1*` and `t2*`, then `t1` and `t2` must be structurally equivalent. Since structs may be nested, and thus multiple structs may start at an address, the function $\text{TYPE}(a)$ returns the set of types starting at address a.

Definition 2. *A stack/global chunk $v \in \mathcal{V}$ is an* **entry point** *if it (a) contains any pointer variable with a target address in the heap or (b) contains a strand cell (e.g., holds the "head" node in a list).*

We begin the formalization of a strand using a pointer that establishes a linkage condition between two cells, see Fig. 3(a) for details in the following. Set operators with a bar, $\bar{\in}$, $\bar{\subseteq}$ and $\bar{\cap}$, function on memory ranges, e.g., $a \mathrel{\bar{\subseteq}} b$ determines if the range of a is included in the range of b.

Definition 3. *A* **cell** *c is a subregion of a memory chunk, i.e., $\exists v \in \mathcal{V} : c \mathrel{\bar{\subseteq}} v$, which begins at address $c.bAddr$ and ends at address $c.eAddr$.*

Definition 4. *A* **linkage condition** *$L = (t, o)$ exists between two cells $c_s \xrightarrow{L} c_t$ with cell type t and linkage offset o if:*

$$\exists(\_, a_s, \_, a_t) \in \mathcal{E} : a_s \mathrel{\bar{\in}} c_s \wedge a_t = c_t.bAddr \wedge o = a_s - c_s.bAddr \\ \wedge\, t \in \text{TYPE}(c_s.bAddr) \cap \text{TYPE}(c_t.bAddr) \wedge c_s \mathrel{\bar{\cap}} c_t = \varnothing.$$

We are interested in the maximal linkage condition, i.e., choose L such that the length of the sequence of cells $c_1 \xrightarrow{L} c_2 \xrightarrow{L} c_3 \ldots$ is maximized. If more than one maximal L exists, then we choose the one with the type t of smallest size.

Definition 5. *A* **strand** *$S = (L, C)$ represents the sequence of cells C captured by a maximal linkage condition L. The cell sequence $C = (C_l, C_c)$ is divided into an optional linear start C_l and an optional cyclic tail C_c, although at least one must be non-empty. When both sequences are non-empty, the following holds:*

$$\forall i \in [1..|C_l|-1] : C_l[i] \xrightarrow{L} C_l[i+1] \;\; \wedge \;\; C_l[|C_l|] \xrightarrow{L} C_c[1]$$
$$\wedge \; \forall i \in [1..|C_c|-1] : C_c[i] \xrightarrow{L} C_c[i+1] \; \wedge \; C_c[|C_c|] \xrightarrow{L} C_c[1].$$

Strands are created to capture every unique sequence of cells and are not destroyed unless all their component cells cease to exist.

As the key to our approach is the reinforcement of evidence via grouping elements that perform the same role, we must ensure that identical strand connections may be found and grouped wherever possible. Thus, due to the issues of Sec. 2, all strand connection parameters (w, x, y and z in the following) are given relative to the cells, linkage pointers and target addresses, i.e., quantities that are independent of a cell's position in a memory chunk (see Figs. 3(b) & (c)). It is for this reason that the strand connections of Fig. 2 are drawn with different line styles, those with the same style have identical parameters. Lastly, note that indirect connections can be generalized to sequences of pointers.

Definition 6. *A* **strand connection** $S_1 \overset{\alpha}{\cdots} S_2$ *describes exactly one way in which a subset of the cells of* S_1 *are related to a subset of the cells of* S_2*. A connection is defined by the cells that establish the relationship:* $\textsc{pairs}(S_1 \overset{\alpha}{\cdots} S_2) = \{(c_1, c_2) \in \textsc{cells}(S_1) \times \textsc{cells}(S_2) : c_1 \overset{\alpha}{\cdots} c_2\}$*. The relationship between cell pairs (and by extension between strands) may be (a) overlay* $c_1 \xleftrightarrow{wx} c_2$ *if* $\textsc{vertex}(c_1) = \textsc{vertex}(c_2)$ *with parameters* $w = (c_2.bAddr + \textsc{linkageOffset}(S_2)) - c_1.bAddr$ *and* $x = (c_1.bAddr + \textsc{linkageOffset}(S_1)) - c_2.bAddr$*. Alternatively, (b) indirect* $c_1 \xrightarrow{yz} c_2$ *if* $\exists e = (v_s, a_s, v_t, a_t) \in \mathcal{E} : v_s \neq v_t \wedge v_s = \textsc{vertex}(c_1) \wedge v_t = \textsc{vertex}(c_2)$ *and there is no linkage condition on* e*. In this case, the parameters are:* $y = a_s - c_1.bAddr$ *and* $z = (c_2.bAddr + \textsc{linkageOffset}(S_2)) - a_t$*.*

To uniquely track the strands reachable from an entry point over multiple time steps, we introduce *entry point connections* for each type of entry point given in Def. 2. These are essentially specialized strand connections, where the starting offset is given from the memory chunk's start address and is therefore absolute. Thus, when a chain of strand connections are followed by their relative offsets, the chain is still uniquely identifiable due to the absolute offset of the initial entry point connection.

Definition 7. *An* **entry point connection** $v_{ep} \xrightarrow{xy} S$ *from an entry point* $v_{ep} \in \mathcal{V}$ *of type Def. 2(a) to a cell* $c \in \textsc{cells}(S)$ *via a non-linkage condition edge* $e = (v_{ep}, a_s, v_t, a_t) \in \mathcal{E}$ *is defined by two parameters:* $x = a_s - v_{ep}.bAddr$ *and* $y = (c.bAddr + \textsc{linkageOffset}(S)) - a_t$*. An entry point connection* $v_{ep} \xrightarrow{z} S$ *from an entry point* $v_{ep} \in \mathcal{V}$ *of type Def. 2(b) to a cell* $c \in \textsc{cells}(S)$ *such that* $c \mathrel{\bar{\subseteq}} v_{ep}$ *is defined by one parameter:* $z = (c.bAddr + \textsc{linkageOffset}(S)) - v_{ep}.bAddr$*.*

Definition 8. *A* **strand graph** $G^s = (\mathcal{V}^s, \mathcal{E}^s)$ *is composed of a vertex set* $v \in \mathcal{V}^s$*, where* v *represents either a strand or an entry point, and an edge set* $e \in \mathcal{E}^s$*, where* e *represents either a strand connection or an entry point connection.*

4.2 Evidence Discovery and Reinforcement

With the strand graph for each time step to hand, we proceed to discover and reinforce evidence for the memory structures of Table 1.

Definition 9. *A* **memory structure** $M = (L, P_{shape}, P_{area}, E)$ *has a label* L*, a shape predicate* P_{shape} *to enable discovery of* L*, an area condition* P_{area} *that describes on which graph elements* P_{shape} *is checked, and an evidence count* E*.*

An area condition serves two purposes, firstly, to limit the number of locations that a shape predicate must be checked, and secondly, to expose to the shape predicate only the subset of graph elements necessary for discovery. Such elements include the set $\mathcal{C}'$, containing the cells of all strands mentioned in the area condition, and the set $\mathcal{E}'$, containing all edges that form the linkage of the strands and all pointers included in strand connections mentioned in the area condition. For Category 1/2 memory structures, P_{area} simply limits whether P_{shape} applies to strands connected by an overlay or an indirect strand connection; however, later we will present a shape predicate employing $\mathcal{C}'$ and $\mathcal{E}'$.

During the construction of the strand graph, for each strand connection matching P_{area}, the associated shape predicate P_{shape} is tested. If found to be true, then the pair (L,E) is added to the strand connection identified by P_{area}. We now give concrete examples of these concepts for selected Category 1/2 memory structures:

$$L = \textsc{sharing}, P_{\text{area}} = S_1 \stackrel{xy}{\longleftrightarrow} S_2 \wedge x = y, E = |\textsc{pairs}(S_1 \stackrel{xy}{\longleftrightarrow} S_2)|, P_{\text{shape}} = \text{true}$$

$$\begin{aligned} L = \textsc{intersectingLists}, P_{\text{area}} &= S_1 \overset{\alpha}{\cdots} S_2, E = |\textsc{pairs}(S_1 \overset{\alpha}{\cdots} S_2)| \\ P_{\text{shape}} &= |\textsc{pairs}(S_1 \overset{\alpha}{\cdots} S_2)| \geq 1 \wedge \neg\textsc{dll}(S_1 \overset{\alpha}{\cdots} S_2) \wedge \neg \ldots \end{aligned}$$

$$\begin{aligned} L = \textsc{dll}, P_{\text{area}} &= S_1 \stackrel{xy}{\longleftrightarrow} S_2, E = |\textsc{pairs}(S_1 \stackrel{xy}{\longleftrightarrow} S_2)| * 3 \\ P_{\text{shape}} &= |\textsc{cells}(S_1)| = |\textsc{cells}(S_2)| = |\textsc{pairs}(S_1 \stackrel{xy}{\longleftrightarrow} S_2)| \\ &\qquad \wedge \textbf{ let } (\_, (C_l^1, C_c^1)) = S_1 \wedge (\_, (C_l^2, C_c^2)) = S_2 \textbf{ in } C_c^1 = C_c^2 = \emptyset \\ &\qquad\quad \wedge \forall i \in [0..\textsc{length}(S_1) - 1]\ \exists (c_1, c_2) \in \textsc{pairs}(S_1 \stackrel{xy}{\longleftrightarrow} S_2) : \\ &\qquad\qquad C_l^1[i+1] = c_1 \wedge C_l^2[\textsc{length}(S_2) - i] = c_2 \end{aligned}$$

Sharing describes two lists that share a downstream cell sequence, while INTERSECTINGLISTS is a catch-all predicate that matches any pair of connected strands. As such, most memory structures must be explicitly excluded in its P_{shape} to prevent unnecessary evidence being produced. While SHARING and INTERSECTINGLISTS generate evidence in the number of connection points between the two strands, the DLL predicate requires two strands to be connected in a specific way and, thus, uses the length of each strand summed with the number of connection points as evidence. Lastly, note that DLL is easily extended to cyclic DLLs by requiring S_1 and S_2 to have cyclic cell sequences and checking that, under some cyclic permutation of those sequences, the DLL property holds.

With the evidence for Category 1/2 memory structures added to the strand graph, we proceed to identify structural repetition via Alg. 1. This serves two

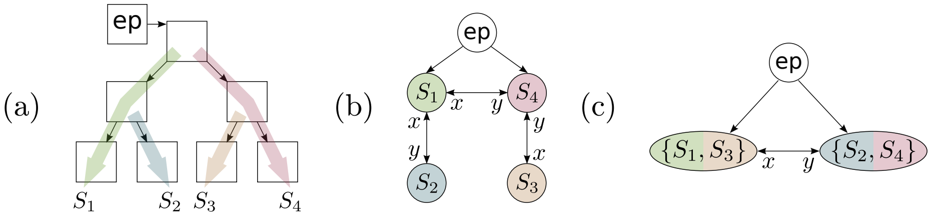


Fig. 4. Tree example: (a) points-to graph, (b) strand graph and (c) folded strand graph.

purposes in our approach. Firstly, it reinforces evidence of Category 1/2 data structures and, thus, alleviates the degenerate shape problem. Secondly, it partially groups the graph elements of Category 2+ data structures, which facilitates their discovery.

Algorithm Sketch 1. Structural repetition *is found by successively locating strands of the strand graph that conceptually perform the same role, and then merging them. Any duplicate strand connections that result from the strand merge are also merged, thus summing any associated evidence. Two strands S_1 and S_2 are merged if they have the same linkage condition L and there exists a merge point S_3 with strand connections $S_3 \overset{\alpha}{\cdots\cdots} S_1 \wedge S_3 \overset{\alpha}{\cdots\cdots} S_2$. A strand connection between S_1 and S_2 is forbidden unless it describes sharing. Alternatively, two strands S_1 and S_2 are merged if they appear in a disjoint partition of the strand graph, where the only connections between strands of that partition describe sharing. After all merges are performed, the result is a folded strand graph.*

Definition 10. *A* **folded strand graph** G^{fs} *is a summarization of a strand graph. The vertices now represent entry points or* sets *of strands. Edges represent entry point connections or* merged *strand connections.*

Area conditions for the discovery of Category 2+ data structures may describe sets of strands, and for convenience these memory structures are discovered in the folded strand graph. Consider the binary tree shown in Fig. 4(a), which has strands covering the left and right linkages; the associated strand graph is shown in Fig. 4(b). This data structure displays high structural repetition and, in the folded strand graph Fig. 4(c), the strands have been grouped into two classes representing the left and right linkages. As can be seen in the following, P_{area} for a binary tree recognizes the shape of Fig. 4(c) exactly:

$$
\begin{aligned}
&L = \textsc{BinaryTree}, P_{\text{area}} = \mathcal{S}_1 \overset{xy}{\longleftrightarrow} \mathcal{S}_2, E = |\textsc{pairs}(\mathcal{S}_1 \overset{xy}{\longleftrightarrow} \mathcal{S}_2)| \\
&\quad P_{\text{shape}} = \textbf{let } \exists \mathcal{E}'_1, \ldots, \mathcal{E}'_n : \mathcal{E}' = \dot\cup_{i=1}^{n} \mathcal{E}'_i \wedge \exists \mathcal{C}'_1, \ldots, \mathcal{C}'_n : \mathcal{C}' = \dot\cup_{i=1}^{n} \mathcal{C}'_i \textbf{ in} \\
&\qquad\qquad \forall i \in 1..n\ \exists c_{\text{root}} \in \mathcal{C}'_i : \\
&\qquad\qquad\qquad (\nexists(\_, \_, \_, a_t) \in \mathcal{E}'_i : a_t \in c_{\text{root}}) \\
&\qquad\qquad\quad\ \wedge |\{(\_, a_s, \_, \_) \in \mathcal{E}'_i : a_s \in c_{\text{root}}\}| \in \{0, 1, 2\} \\
&\qquad\qquad\quad\ \wedge \forall c \in \mathcal{C}'_i - c_{\text{root}} : |\{(\_, \_, \_, a_t) \in \mathcal{E}'_i : a_t \in c\}| = 1 \\
&\qquad\qquad\qquad\ \wedge |\{(\_, a_s, \_, a_t) \in \mathcal{E}'_i : a_s \in c \wedge a_t \in \mathcal{C}'_i - \{c_{\text{root}}\}| \in \{0, 1, 2\}
\end{aligned}
$$

The shape predicate employs the sets $\mathcal{C}'$ and $\mathcal{E}'$ that result from P_{area} to ensure that irrelevant pointers are excluded from the shape test. However, due to the folding of structural repetition, it is possible that P_{area} locates multiple trees. This could occur if, e.g., many trees were nested under an SLL. To handle this, P_{shape} first partitions $\mathcal{C}'$ and $\mathcal{E}'$ into n trees using the disjoint union operator $\uplus$, where $\mathcal{C}'_i$ and $\mathcal{E}'_i$ represent the elements of tree i. Then, for each i, a root c_{root} is found with no incoming pointer in $\mathcal{E}'_i$, and the non-root cells $\mathcal{C}'_i - c_{\text{root}}$ are checked for a suitable number of incoming and outgoing edges in $\mathcal{E}'_i$.

If P_{shape} is found to be true for a Category 2+ data structure, then all strand connections mentioned in P_{area} have (L, E) added. Since the label and evidence may be distributed over multiple elements of the folded strand graph, L is parameterized to ensure the graph elements of that memory structure can be recovered. However, for a binary tree such a parameterization is unnecessary.

To find temporal repetition we must locate strands that perform the same role over multiple time steps. As mentioned previously, we do not attempt a global solution and instead solve the problem from the point of view of each entry point, where that local solution is represented as follows:

Definition 11. *An* **aggregate strand graph** G^{as}_{ep} *is composed of edges describing strand connections and vertices, of which one, v_{ep}, will represent the entry point and the remainder will represent linkage conditions.*

Algorithm Sketch 2. Temporal repetition *observed by an entry point* `ep` *is computed as follows. For each time step t in* `ep`*'s lifetime, we extract the subgraph of G^{fs}_t reachable from v_{ep}, which results in a subgraph set $\mathcal{G}$. To abstract over multiple time steps, we relabel all vertices that represent strands in the graphs of $\mathcal{G}$ to include only the associated linkage condition, which, unlike strands, is time step independent. The subgraphs in $\mathcal{G}$ are merged together in time step sequence, where the result of the last merge G^{as}_{ep} is merged with the next subgraph $G_{next} \in \mathcal{G}$ and G^{as}_{ep} is initially empty.*

To perform the merge, v_{ep} of each graph is placed in correspondence, and then an inexact graph match is computed. Vertices may be in correspondence if they have identical linkage conditions, while edges may be in correspondence if the strand connections (including parameters) are identical. Graph elements in correspondence imply that temporal repetition has been discovered, and naturally the merge also sums any associated evidence. Elements of G_{next} not in correspondence are simply transfered with their associated evidence to G^{as}_{ep}.

The identification algorithm is then applied to each G^{as}_{ep}, resulting in a natural language string. However, due to space limitations we refer the reader to the informal description of this process presented at the end of Sec. 3.

5 Preliminary Results

We have prototyped our approach using a combination of CIL [10] (approx. 1K LOC OCaml & 600 LOC C) to inject instrumentation into C source code, and

Table 2. Preliminary results obtained from our prototype implementation.

Example	Runtime (s)	Memory (GB)	Evidence Count	% Supp. / % Opp.	# Agg. Merges
Binary Tree	10.3	2.70	**Tree**: 102, Nesting: 21	83%/17%	16
Linux DLL [1]	9.7	1.76	**CDLL**: 60, IL: 52, DLL: 6	51%/49%	20
Wolf DLL [12]	71.4	2.86	**DLL**: 1410, IL: 220	87%/13%	123
Skip list with DLL Children	107.2	2.84	**SL**: 24793, N: 487, Tree: 48	98%/2%	101
			DLL: 345, IL: 2	99%/1%	

Scala (approx. 7.5K LOC) to perform the offline analysis. All experiments were run on an Intel i7-4800MQ with 32GB of RAM. We applied the prototype to four examples, the first three of which are self-written: a binary tree, an example exercising the cyclic Linux DLL [1], a skip list with child DLLs and a textbook DLL implementation [12]. We have made the source code of our self-written examples available at `http://www.david-white.net/kps15.zip`.

In Table 2 we report the runtime and memory usage of the offline analysis, although we note that currently no optimization has been performed and we store much redundant data. To simplify presentation of the results, we give details for only the longest running entry point of each example. In the evidence column we list all the non-zero evidence counts for each discovered memory structure, and in the following column we give the ratio of evidence supporting the correct data structure name versus that opposing. In each example, the evidence suggests the correct memory structure, which is shown in bold. For the example with multiple data structures, we separate out the evidence for each sub structure into a separate row. Lastly, in column *# Agg. Merges*, we show the number of merges that were performed by Algorithm 2 to produce the aggregate strand graph, which gives a rough indication of the rate at which evidence was gathered.

All examples are synthetic in the sense that they only manipulate, often just one, data structure. Since real world programs perform other tasks besides data structure manipulation, their data structures typically spend a smaller proportion of the runtime in degenerate states. Therefore, although the examples are quite simple, they effectively represent the worst case for our evidence based analysis. This is especially true for the Linux DLL example, which has approximately three time steps of intersecting lists for every one time step where the full DLL is present, resulting in only 51% evidence supporting a DLL. Behavior more typical of a real-world program can be seen in the skip list with nested DLLs example. This is due to building the skip list first, which is then held in a stable shape while child DLLs are added. This stable portion generates overwhelming evidence for the skip list.

6 Conclusion

We have presented dsOli2, a dynamic analysis that automatically identifies the dynamic data structures appearing in a C program during execution. By decomposing complex structures into strands and then analyzing the resulting strand connections, we are able to identify many data structures typically appearing in C heaps such as (cyclic) singly and doubly linked lists, trees, skip lists and relationships between data structures such as nesting. In contrast to related work that tries to avoid degenerate shapes [11, 7, 5], we permit these in our analysis and employ evidence based on structural complexity that is reinforced by structurally and temporally repetitive heap structures to override degenerate shapes. Preliminary results appear to support this method of dealing with degenerate shapes, and does not require the discovery of data structure operations or quiescent periods.

Ultimately, we aim to use the output of dsOli2 in a number of applications beyond program comprehension, including informing formal verification (which has already been studied in the context of dsOli1 and VeriFast [9]) and reverse engineering, which would be possible after we permit object code as input.

Acknowledgments. This work is supported by DFG grant LU 1748/4-1.

References

1. Linux kernel 4.1 cyclic dll (`include/linux/list.h`). `http://www.kernel.org/`. Accessed: 31/08/2015.
2. Braginsky, A. and Petrank, E. Locality-Conscious Lock-Free Linked Lists. In *ICDCN 2011*, vol. 6522 of *LNCS*, pp. 107–118. Springer, 2011.
3. Caballero, J., Grieco, G., Marron, M., Lin, Z., and Urbina, D. ARTISTE: Automatic Generation of Hybrid Data Structure Signatures from Binary Code Executions. Tech. Report TR-IMDEA-SW-2012-001, IMDEA Software Institute, 2012.
4. Dudka, K., Peringer, P., and Vojnar, T. Byte-Precise Verification of Low-Level List Manipulation. In *SAS 2013*, vol. 7935 of *LNCS*, pp. 215–237. Springer, 2013.
5. Haller, I., Slowinska, A., and Bos, H. MemPick: High-Level Data Structure Detection in C/C++ Binaries. In *WCRE 2013*, pp. 32–41. IEEE, 2013.
6. Holík, L., Lengál, O., Rogalewicz, A., Šimáček, J., and Vojnar, T. Fully Automated Shape Analysis Based on Forest Automata. In *CAV 2013*, vol. 8044 of *LNCS*, pp. 740–755. Springer, 2013.
7. Jung, C. and Clark, N. DDT: Design and Evaluation of a Dynamic Program Analysis for Optimizing Data Structure Usage. In *MICRO 2009*, pp. 56–66. IEEE, 2009.
8. Marron, M., Sanchez, C., Su, Z., and Fahndrich, M. Abstracting Runtime Heaps for Program Understanding. *IEEE Trans. Softw. Eng.*, 39(6):774–786, 2013.
9. Mühlberg, J. T., White, D. H., Dodds, M., Lüttgen, G., and Piessens, F. Learning Assertions to Verify Linked-List Programs. *To appear at the 13th International Conference on Software Engineering and Formal Methods*, 2015.
10. Necula, G. C., McPeak, S., Rahul, S. P., and Weimer, W. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *CC 2002*, vol. 2304 of *LNCS*, pp. 213–228. Springer, 2002.
11. White, D. H. and Lüttgen, G. Identifying Dynamic Data Structures by Learning Evolving Patterns in Memory. In *TACAS 2013*, vol. 7795 of *LNCS*, pp. 354–369. Springer, 2013.
12. Wolf, J. *C von A bis Z.* Galileo Computing, 2009.

Comprehensive Multi-platform Dynamic Program Analysis for Java and Android

Yudi Zheng[1], Stephen Kell[2], Lubomir Bulej[1], Haiyang Sun[1], and Walter Binder[1]

[1] Università della Svizzera italiana (USI)
{yudi.zheng, haiyang.sun, lubomir.bulej, walter.binder}@usi.ch

[2] University of Cambridge
stephen.kell@cl.cam.ac.uk

Abstract. Dynamic program analyses, such as profiling, tracing and bug-finding tools, are essential for software engineering. Unfortunately, implementing dynamic analyses for managed languages such as Java is unduly difficult and error-prone, because the runtime environments provide only complex low-level mechanisms. Currently, programmers writing custom tooling must expend great effort in tool development and maintenance, while still suffering substantial limitations such as incomplete code coverage or lack of portability. Ideally, a framework would be available in which dynamic analysis tools could be expressed at a high level, robustly, with high coverage and supporting alternative runtimes such as Android. We describe our research on an "all-in-one" dynamic program analysis framework which uses a combination of techniques to satisfy these requirements.

Keywords: Dynamic program analysis, Java, Android

1 Introduction

Have you ever wanted to climb inside your program to see it executing? Modern, managed platforms such as the Java Virtual Machine (JVM) expose a variety of low-level interfaces for instrumenting and profiling code, but obtaining high-level insight remains frustratingly difficult.

Developers of large, complex systems have a continual need to optimize, test, debug and comprehend their systems' behavior. For example, when investigating performance, we might want to count objects allocated by allocation site (allocation profiling), log entry and exit to certain methods (method tracing), count caller–callee invocation frequencies (call-graph edge profiling), flag lines of code as covered or not (code coverage), and so on. These are all *dynamic program analyses*, offered by various off-the-shelf tools. Since complex programs vary in what methods are of interest, how allocation sites should be grouped together, how much context sensitivity is appropriate, and so on, programmers often require more tailored analyses. Therefore programmers nevertheless frequently customize their tooling, by grappling with the VM's low-level interfaces.

The basic such interface offered by a JVM is *bytecode instrumentation*. Using an API called JVMTI [12] and a bytecode library such as ASM (`http://asm.ow2.org`), the tool author rewrites the program's assembly-level bytecode instructions as they are loaded. This is intricate and error-prone: it must add analysis logic, but otherwise avoid interfering with the program's execution. It's also insufficient: some events (e.g. object allocation) occur not only in bytecode but also internally within the VM, requiring a separate set of callbacks. Using JVMTI is both difficult and commonplace, as revealed by hundreds of Stack Overflow questions.

Bytecode instrumentation has the appealing property that the analysis and the program share a virtual machine. The core of the analysis can therefore be written in Java or another familiar language, and is dynamically optimized together with the program. Unfortunately, this also creates a fundamental tension between *coverage* and *isolation*. The analysis inevitably interferes with the program's behavior, since it shares the same core classes. The consequences range from the typically harmless (class initializers run in a different order after instrumentation) to the surprisingly deadly: infinite recursion, state corruption or deadlock. The usual escape route is to leave core libraries uninstrumented, sacrificing coverage.

Is there a better way? Ideally, we would like a high-level programming model that abstracts away from bytecode. We would also like high coverage, allowing the instrumentation of core classes without risk of interference. The analysis should also be portable to any JVM and perhaps other VMs such as Dalvik (used in the Android operating system).

Our research has produced an "all-in-one" analysis framework that achieves these goals. As we'll see, it comprehensively takes care of the incidental complexities of developing custom dynamic analyses, allowing programmers to focus on the essentials.

2 Writing Dynamic Analyses is Hard

Let's examine a real-world example. JaCoCo [5] is a code coverage tool reporting which classes, methods and lines of code were touched during a given program execution. It maintains arrays of flags on a per-class basis, and instruments application code to set flags as control reaches the corresponding points. This is easy to state, but not easy to implement: JaCoCo's core and runtime implementation amounts to about 2000 logical lines of Java. Much of this code is devoted to manipulating bytecode instructions. Mixed in with this is the primary concern of creating and updating the arrays.

The extract in Figure 1 shows the kind of code involved. Instrumentation is done using the ASM bytecode library. Similar libraries include Shrike (`http://wala.sourceforge.net/wiki/index.php/Shrike_technical_overview`), which offers a patch-like abstraction on bytecode, and Javassist [4], which integrates into the Java class-loading infrastructure.

The intention of this code is simple: getting a local reference to a system-wide array of flags corresponding to the lines of code covered. The array is retrieved via Java's system properties object; notice how a canned bytecode sequence for

```
// in SystemPropertiesRuntime
public int generateDataAccessor( final long classid , final String classname,
    final int probecount, final MethodVisitor mv) {
  mv.visitMethodInsn(Opcodes.INVOKESTATIC, "java/lang/System",
      " getProperties" , " ()Ljava/ util / Properties ;" , false );

  // Stack [0]: Ljava/ util / Properties ;

  mv. visitLdcInsn (key );

  // Stack [1]: Ljava/lang/String ;
  // Stack [0]: Ljava/ util / Properties ;

  mv.visitMethodInsn(Opcodes.INVOKEVIRTUAL, "java/util/Properties",
      "get" , "(Ljava/lang/Object;)Ljava/lang/Object;" , false );

  // Stack [0]: Ljava/lang/Object;

  RuntimeData.generateAccessCall( classid , classname, probecount, mv);

  // Stack [0]: [Z

  return 6; // Maximum local stack size is 3
}
```

Fig. 1: Direct bytecode instrumentation in JaCoCo [5]

calling System.getProperty() is spliced in by manually assembling bytecode ("visit" means "append instruction to the output buffer") and explicitly managing the operand stack.

We can also see the potential for interference problems. The library method System.getProperties() might itself be instrumented. To avoid infinite recursion, we need to arrange for the instrumentation to call an *uninstrumented* version of it. Alternatively, we could exclude the method from instrumentation entirely (as is done by JaCoCo), but then we would not measure its coverage. In general, this sharing of library state between program and instrumentation risks modifying the program behavior in unforeseeable ways, depending on the internals of the library [6].

These difficulties motivate a different approach. Developing a dynamic analysis involves writing two different kinds of code. Some code does *instrumentation*—inserting logic into the base program, to collect low-level observations. Other code does *analysis*, turning these observations into the high-level output desired by the user. In most cases, the inserted code is simple: it collects contextual information at the insertion site (e.g., the index of the bytecode instruction that has been hit, which class and method it is in, etc.). By contrast, the analysis might perform complex computations to aggregate and filter the output.

Ideally, therefore, analyses would be written in an ordinary, powerful, general-purpose programming language. Instrumentation, by contrast, inserts only simple code, but requires some specialized notation to specify *what* information to collect and *when*. Mixing instrumentation and analysis tends to make both kinds of code unnecessarily complex [1]. In our example, the array retrieved by the getProperties() call in Figure 1 is really part of the analysis—it is used to aggregate code coverage events—yet is being dealt with by instrumentation. We would like a design that keeps the two separate.

Although the inserted code is simple, inserting it is not. This is a problem of meta-programming—modifying the structure of another program. It must transform arbitrary bytecode to collect the required information (*what*) at the required points (*where*) while otherwise faithfully preserving its semantics. Normally, instrumentation is viewed as a special case of program transformation, and programmed by manipulating free-form lists of instructions. Although flexible, this is needlessly onerous, since instrumentation seeks only to *add* behavior, not modify it. Rather than manipulating raw instructions, we require a carefully designed set of primitives which express *addition of code* straightforwardly.

We find inspiration for these primitives in aspect-oriented programming (AOP) [7], and its notions of *join points* (dynamic points in execution) and *advice* (code snippets inserted into existing code). It is possible to use an existing aspect-oriented language like AspectJ for some instrumentation tasks, but this suffers numerous limitations: AspectJ cannot instrument core library classes (conservatively avoiding interference problems) and lacks definitions for many of the intra-procedural control-flow join points commonly used in analyses, such as basic block entry/exit.

If we specify instrumentation using aspect-like primitives, how does this integrate with the analysis code? One way is to treat a dynamic analysis as a (potentially distributed) event-processing system. This decouples the two kinds of code, and abstracts away from instrumentation mechanisms. There is a natural mapping from event-processing concepts onto dynamic program analysis.

Events. Events reify specific moments in the execution of the base program, along with relevant contextual information. Events are produced by instrumentation and consumed by analysis.

Producers. An event producer is a unifying abstraction of various program instrumentation mechanisms. For example, on the JVM we have two mechanisms: bytecode instrumentation and JVMTI agent callbacks [12].

Consumers. An event consumer is a unifying abstraction of analysis code. An analysis specifies only which events it requires, not how they are collected. It consumes these events and generates output useful to the application developer.

3 The ShadowVM Framework

The ShadowVM framework is the "all in one" system we have built to implement our vision of simple custom dynamic analyses. It lets developers retain Java as the primary development language. By separating instrumentation from analysis, it offers a higher level of abstraction than bytecode instrumentation. Figure 2 illustrates how it realizes dynamic analyses as distributed event-processing systems. The base program executes in the *observed VM*, where instrumentation produces events. The framework delivers these to the analysis, executing in the separate *ShadowVM*.

Producer programming model. In the observed VM, instrumentation produces events that are required by the analysis. We adopt the aspect-oriented programming model of DiSL, a domain-specific language embedded in Java [9]. It expresses instrumentation using the abstractions of *markers*, *guards* and *snippets*. Markers identify points in execution, which guards may filter. Snippets, analogous to *advice* in AOP, are small fragments of Java code targeting the *Event API*. This API accepts events for delivery to the analysis.

Events may be constructed from primitive values, strings, object identities, and a selection of data types identifying locations in code: classes, method names, and marker-defined identifiers such as basic block IDs. A library of ready-made markers and snippets is provided to generate common bytecode events, such as method entry/exit, basic-block entry/exit, object allocation, or field read/write.

VM-internal events, not corresponding to bytecodes, are denoted by the unit of *resource* whose lifetime they relate to: objects, threads, or the VM itself. The framework generates events marking the disposal of resources, often useful as triggers for analyses to clean up internal state or output results.

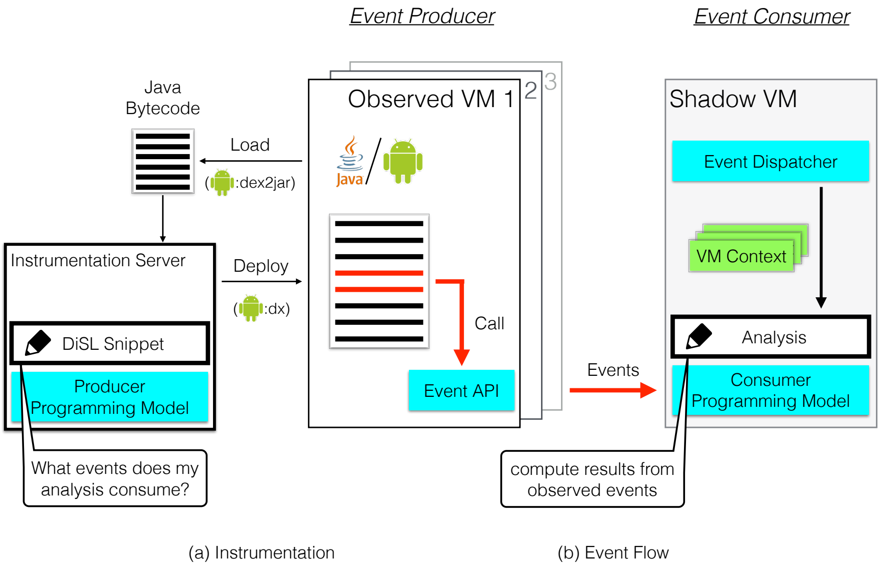


Fig. 2: Overview of the ShadowVM framework.

Consumer programming model. All analysis state and computation occurs in the ShadowVM, using facilities of the *shadow API* [8, 14]. Its basic abstraction is the *shadow object*. Logically, any object in the base program has a corresponding shadow in the analysis. In practice, shadows are created on demand. When an object is first passed to the *Event API*, it is tagged with a unique 64-bit number, and a shadow object is created, recording this identifier and the base object's class. Beyond this, shadow objects' state is user-defined, consisting of an arbitrary key-value map. This may be used to store analysis-specific data (e.g. timestamps, flags, etc.) and/or the real object's contents (by observing field writes; library code is provided). Many performance-oriented analyses do not require object contents. Shadow strings are a special case: for convenience, they replicate the base string contents.

Event notifications are delivered as method invocations on an analysis class loaded in the ShadowVM, somewhat similar to remote method invocation. The analysis developer controls the interface of this class, so each kind of event corresponds to a method of specific signature. Generally, the developer supplies instrumentation, typically chosen from a library, to generate these events. In the case of lifetime events, the developer simply implements a system-defined interface corresponding to the desired kinds of lifetime events—object death, thread termination, or exit of the observed VM—signaling to the framework that it must generate these events. (On the JVM, these events are generated by registering JVMTI callbacks.)

Configuration issues. For use cases where only specific packages must be instrumented, the developer may define a "scope" (set of classes to instrument) and/or a global exclusion list. Wildcards are supported, e.g., `exclude "java.*"`. In the absence of these, all bytecode is subject to instrumentation; unlike other systems, our system safely supports this. Additionally, each instrumentation can be guarded by conditions that are evaluated at instrumentation time, referring to any property of the class/method being instrumented.

4 Supporting the Android Platform

Android is a Linux-based multi-user operating system. Applications are written in Java, and executed in the Dalvik Virtual Machine (DVM). The DVM lacks certain features that enable implementation of the ShadowVM framework on the JVM, most notably a tool interface akin to JVMTI. Extending the ShadowVM to support Android therefore required overcoming various conceptual and technical challenges.

Multi-process application support. Although written in Java, Android applications adhere to a particular component model, and expose multiple entry points. By default, the components of a single application execute in a single DVM, but any component can be configured to execute in a separate DVM, distributing the application across address spaces. An analysis observing an Android application therefore needs to handle events from multiple VM instances. ShadowVM enables this by associating the observed events and object identities with a "VM context" provided to the analysis with each delivered event (Figure 2b). New DVM instances are spawned from a bootstrap VM (the *Zygote*), requiring the ShadowVM to replicate shadow objects from the zygote's shadow into its new child. Replication of any custom data associated with shadow objects in the parent VM is handled by the analysis, but this only concerns objects that were exposed to the analysis during initialization of the system classes in the *Zygote*.

Inter-process communication events. Android applications execute in a private sandbox. Each application has its own data, and can communicate and exchange data with other applications or services through the *Binder* inter-process communication (IPC) mechanism. The communication follows a synchronous client-server model, transferring control flow between client and server with each request and response. To enable observation of multi-process applications and their interactions with the wider system, the ShadowVM framework on Android expands the range of VM-internal events to include the low-level IPC operations that Android applications use for communication and control transfer.

Tool interface essentials. On the JVM, JVMTI is used to instrument classes on load and to implement the generation of VM-internal object, thread, and VM lifecycle events. DVM lacks any similar tool interface, so we needed to modify the DVM to provide the essential subset of JVMTI features. This includes object tagging, hooks in the garbage collector (when freeing tagged objects) and in various other places (e.g., class loading, IPC, thread creation and termination, etc.). Our modifications to DVM are encapsulated in well-defined interfaces, making them portable to the new Android Runtime (ART, from the recent Android 5.0 release).

Bytecode transformation and class loading. The DVM implements a register-based machine, and works with bytecode converted from the stack-based Java bytecode. Working directly with Dalvik bytecode would place an added burden on analysis developers, requiring platform-specific instrumentations to enable development of multi-platform analyses. We avoid this by converting the Dalvik bytecode to Java bytecode for instrumentation, and converting it back for execution (Figure 2a). Unlike the JVM, which loads individual classes as streams of bytes, the DVM loads multiple classes at a time by mapping a class archive directly into memory. This forces us to instrument classes in batches before they are mapped into memory, to preserve transparency of load-time instrumentation.

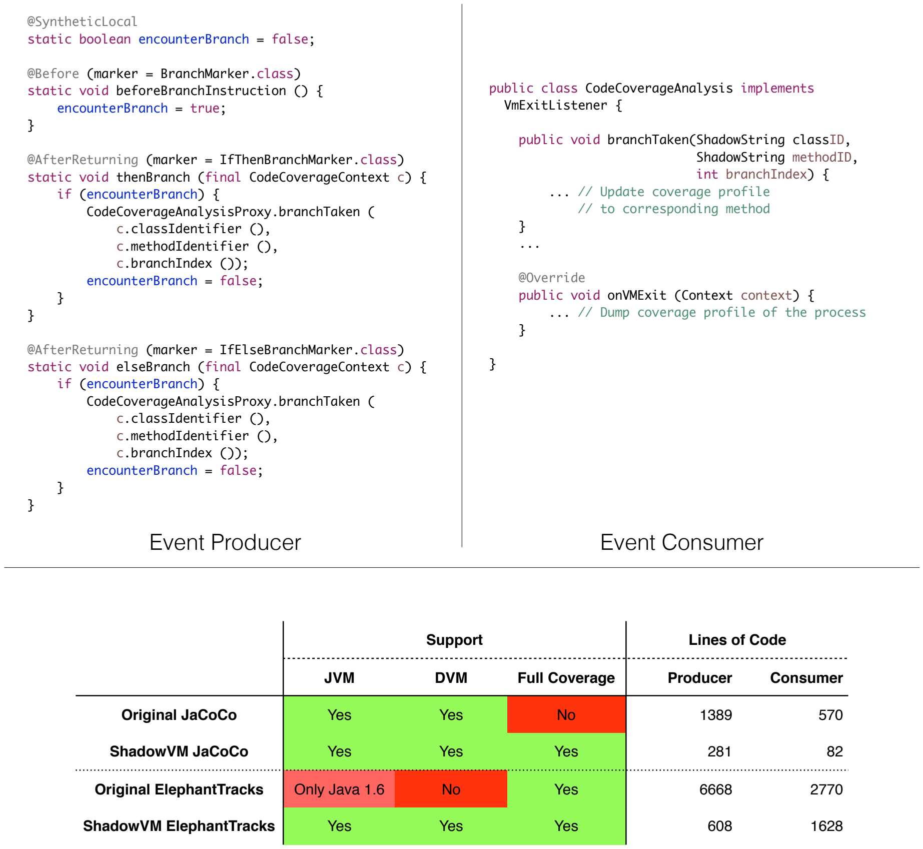
5 Example Analyses

To illustrate the framework, we implemented the functionality of the popular code coverage tool JaCoCo [5] using ShadowVM. The upper part of Figure 3 shows the code snippets for branch event producer (instrumentation) and branch event consumer (analysis). The instrumentation assigns each branch a dedicated number for indexing, and emits an event indicating which branch is taken. This code illustrates our aspect-oriented primitives: Java attributes mark a snippet (a static method) with places where it should be inserted (here before and after branches). The extra “synthetic” local boolean is inserted into each method body and used to select only the taken branches. Although snippets appear as static methods within a Java class, this is simply a convenient container for annotated fragments of code and auxiliary definitions (like the synthetic local). It is never loaded nor instantiated, and is used only by the instrumentation engine.

The snippet produces an event consisting of a string and an integer, uniquely identifying the branch. The analysis maintains a simple data structure tracking taken branches, updated in reaction to the events received.

The bottom part of Figure 3 compares the original JaCoCo with the ShadowVM version. While both versions support the JVM and the DVM, only the ShadowVM version allows code coverage analysis of core library classes. Moreover, our framework enables a more compact implementation of both instrumentation and analysis; overall, the ShadowVM version has fewer than 19% of the logical lines of code of the original JaCoCo.

We also implemented the object-lifetime analyzer ElephantTracks [13] with ShadowVM. The original ElephantTracks is implemented as a native JVMTI agent in C to avoid interference. With ShadowVM, the tool can be implemented in pure Java. The original ElephantTracks only runs on Java 6, whereas the ShadowVM version also supports Java 7, Java 8, and the DVM. Overall, the ShadowVM version has fewer than 24% of the logical lines of code of the original ElephantTracks.



```
@SyntheticLocal
static boolean encounterBranch = false;

@Before (marker = BranchMarker.class)
static void beforeBranchInstruction () {
    encounterBranch = true;
}

@AfterReturning (marker = IfThenBranchMarker.class)
static void thenBranch (final CodeCoverageContext c) {
    if (encounterBranch) {
        CodeCoverageAnalysisProxy.branchTaken (
            c.classIdentifier (),
            c.methodIdentifier (),
            c.branchIndex ());
        encounterBranch = false;
    }
}

@AfterReturning (marker = IfElseBranchMarker.class)
static void elseBranch (final CodeCoverageContext c) {
    if (encounterBranch) {
        CodeCoverageAnalysisProxy.branchTaken (
            c.classIdentifier (),
            c.methodIdentifier (),
            c.branchIndex ());
        encounterBranch = false;
    }
}
```

Event Producer

```
public class CodeCoverageAnalysis implements
  VmExitListener {

    public void branchTaken(ShadowString classID,
                            ShadowString methodID,
                            int branchIndex) {
        ... // Update coverage profile
            // to corresponding method
    }
    ...

    @Override
    public void onVMExit (Context context) {
        ... // Dump coverage profile of the process
    }

}
```

Event Consumer

	Support			Lines of Code	
	JVM	DVM	Full Coverage	Producer	Consumer
Original JaCoCo	Yes	Yes	No	1389	570
ShadowVM JaCoCo	Yes	Yes	Yes	281	82
Original ElephantTracks	Only Java 1.6	No	Yes	6668	2770
ShadowVM ElephantTracks	Yes	Yes	Yes	608	1628

Fig. 3: Top: JaCoCo on ShadowVM; bottom: original JaCoCo and ElephantTracks versus our implementations on ShadowVM

In summary, ShadowVM reduces development effort for many analysis tools, thanks to its high-level programming model, multi-platform support and built-in comprehensive bytecode coverage.

6 Discussion

Any practical system makes certain trade-offs in its design and implementation. We conclude this paper with a discussion of the strengths and limitations of our framework.

6.1 Benefits and Deployment Scenarios

Expressiveness, isolation and complete bytecode coverage. Our approach satisfies the goals we identified at the start of the paper. It offers a favorable trade-off between a high-level programming model and expressiveness. By deploying the analysis in a separate process, interference with the observed application is minimized, and analyses can observe code in core classes, right from the earliest “bootstrapping” stages of VM execution.

Multi-platform analysis. With our framework, all user code is portable: an analysis written for Java applications also supports Android applications out-of-the-box. Our framework also offers multi-process support, such that one analysis process can handle the events of multiple observed JVMs and DVMs. This provides a sound basis for analyzing distributed systems.

Parallelism and available resources. Because the event-consuming part of an analysis executes in a separate VM, our framework implicitly parallelizes the execution of the observed application and the analysis. Since the analysis VM can be deployed on a different machine than the observed VM, our approach minimizes the extra memory requirements on the observed VM. This enables heavyweight analysis even on resource-constrained devices.

6.2 Limitations

Users of our dynamic program analysis framework need to be aware of the following limitations, which our ongoing research is addressing.

Overhead. Since the event-producing and event-consuming parts of an analysis are separate processes, some communication overheads are incurred. We refer to [8] for a performance evaluation of our framework. The implementation of *object tagging* in standard JVMs proves a bottleneck; this is used to assign globally unique identities to objects that are captured in events, and is stressed heavily by our system.

DVM quirks. For the analysis of Android applications, a version-specific patch needs to be applied to the DVM first. The conversion between JVM and Dalvik bytecode introduces some bias in metrics related to individual bytecodes or basic blocks. For example, the basic block size may change upon bytecode conversion.

Event ordering. Concurrency raises some subtle issues in event processing. Our framework supports different event ordering semantics (particularly, global order and per-thread program order [8]), to cater to different analyses’ requirements. However, it does not guarantee that the occurrence of an event and the generation of the event happen atomically. Consequently, the happens-before relationship within an observed VM may not always be preserved. Other program analysis frameworks suffer from the same limitations, independently of whether they perform the analysis within the observed VM or in a separate process.

Native code. Our system cannot observe execution in native code, unlike whole-program dynamic instrumentation systems such as Valgrind [11] or DynamoRIO [2]. These offer instrumentation interfaces at the level of portable intermediate code—much lower-level than our approach. Also, they provide no way to recover a source-level view of a Java program's state in terms of objects, fields, methods, etc.. Systems such as DTrace [3], which instrument native code at both user and kernel levels, face in-kernel isolation problems analogous to those we face in the observed VM. The solutions are similar: to avoid interference, DTrace instrumentation traps to a wait-free code path (analogous to our snippets) which buffers data (events) for hand-off to a sandboxed consumer (the analysis), while sharing no state with the rest of the kernel.

Synchronous analysis. In our system, events are processed remotely and asynchronously by the analysis. This gives the analysis no opportunity to "go back" and inspect more program state than it was initially passed. Instead, all the required state must be captured up-front in the instrumentation. Thus, ShadowVM is not suited for implementing interactive debuggers. Furthermore, the analysis cannot synchronously request a heap dump. We could request the heap dump only later, when it may not show the relevant features. Alternatively, we could maintain a "shadow heap" in the analysis, by observing all events that change the object graph (i.e., field writes). However, this usually exhibits high overhead. Also, changes to the object graph in native code (e.g., through the JNI, upon object cloning, or upon deserialization) are not captured by our current implementation, meaning the shadow heap may not be completely accurate.

6.3 Availability

Our program analysis framework is public available as an open-source software project hosted on OW2 (`http://disl.ow2.org/`). The current release DiSL 2.1 includes both DiSL and ShadowVM; it has been endorsed by the SPEC Research Group and included in their tool repository (`http://research.spec.org/tools/overview/disl.html`). A detailed tutorial helps getting started with DiSL [10]. DVM support is currently available as a prototype (`http://goo.gl/LKmVxf`); it will be part of the forthcoming DiSL 3.0 open-source release.

Acknowledgments

The research presented in this paper has been supported by the Swiss National Science Foundation (projects CRSII2_136225 and 200021_141002), by the European Commission (contract ACP2-GA-2013-605442) and by Oracle (ERO project 1332).

Bibliography

[1] Ansaloni, D., Kell, S., Zheng, Y., Bulej, L., Binder, W., Tůma, P.: Enabling modularity and re-use in dynamic program analysis tools for the Java virtual machine. In: ECOOP '13: Proceedings of the 27th European Conference on Object-Oriented Programming. LNCS, vol. 7920, pp. 352–377 (2013)

[2] Bruening, D., Zhao, Q., Amarasinghe, S.: Transparent dynamic instrumentation. In: VEE '12: Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments. pp. 133–144 (2012)

[3] Cantrill, B.M., Shapiro, M.W., Leventhal, A.H.: Dynamic instrumentation of production systems. In: ATEC '04: Proceedings of the USENIX Annual Technical Conference. pp. 2–15 (2004)

[4] Chiba, S.: Load-time structural reflection in Java. In: ECOOP '00: Proceedings of the 14th European Conference on Object-Orientd Programming. LNCS, vol. 1850, pp. 313–336 (2000)

[5] EclEmma: JaCoCo Java Code Coverage Library, uRL: `http://www.eclemma.org/jacoco/`

[6] Kell, S., Ansaloni, D., Binder, W., Marek, L.: The JVM is not observable enough (and what to do about it). In: VMIL '12: Proceedings of the 6th ACM Workshop on Virtual Machines and Intermediate Languages. pp. 33–38 (2012)

[7] Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: ECOOP '97: Proceedings of the 11th European Conference on Object-Oriented Programming. LNCS, vol. 1241, pp. 220–242 (1997)

[8] Marek, L., Kell, S., Zheng, Y., Bulej, L., Binder, W., Tůma, P., Ansaloni, D., Sarimbekov, A., Sewe, A.: ShadowVM: Robust and comprehensive dynamic program analysis for the Java platform. In: GPCE '13: Proceedings of the 12th International Conference on Generative Programming: Concepts and Experiences. pp. 105–114 (2013)

[9] Marek, L., Villazón, A., Zheng, Y., Ansaloni, D., Binder, W., Qi, Z.: DiSL: A domain-specific language for bytecode instrumentation. In: AOSD '12: Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development. pp. 239–250 (2012)

[10] Marek, L., Zheng, Y., Ansaloni, D., Bulej, L., Sarimbekov, A., Binder, W., Tůma, P.: Introduction to dynamic program analysis with DiSL. Science of Computer Programming 98, part 1, 100–115 (2015)

[11] Nethercote, N., Seward, J.: Valgrind: A framework for heavyweight dynamic binary instrumentation. SIGPLAN Not. 42(6), 89–100 (2007)

[12] Oracle: JVM Tool Interface (JVMTI) Version 1.2, uRL: `http://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html`

[13] Ricci, N.P., Guyer, S.Z., Moss, J.E.B.: Elephant Tracks: Portable production of complete and precise GC traces. In: Proceedings of the 2013 International Symposium on Memory Management. pp. 109–118 (2013)

[14] Sun, H., Zheng, Y., Bulej, L., Villazón, A., Qi, Z., Tůma, P., Binder, W.: A programming model and framework for comprehensive dynamic analysis on Android. In: MODULARITY '15: Proceedings of the 14th International Conference on Modularity. pp. 133–145 (2015)