University Halle-Wittenberg
Institute of Computer Science

Tagungsband 17. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS'13)

Roswitha Picht, Wolf Zimmermann (Hrsg.)

Technical Report 2014/02

University Halle-Wittenberg
Institute of Computer Science

Tagungsband 17. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS'13)

Roswitha Picht, Wolf Zimmermann (Hrsg.)

April 2014 **Technical Report 2014/02**

Institute of Computer Science
Faculty of Natural Sciences III
Martin-Luther-University Halle-Wittenberg
D-06099 Halle, Germany

WWW: http://www.informatik.uni-halle.de/preprints

© All rights reserved.

LaTeX-Style: designed by Winfried Geis, Thomas Merkle, University Stuttgart
adapted by Paul Molitor, Halle
Permit granted by University Stuttgart [25/05/2007]

Tagungsband 17. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS'13)

Roswitha Picht, Wolf Zimmermann (Hrsg.)

Inhaltsverzeichnis

Abstract/Zusammenfassung

Vorwort

Beiträge

pylibjit: A JIT Compiler Library for Python 9
Gergö Barany

Konsistenzprüfung bei der Anforderungsfindung für hydromechatronische Systeme 19
Christian Berg

COMPOSITA: A Study in Runtime Architectures for Massively Parallel Systems 31
Luc Bläser

Ein auf partieller Auswertung basierendes Tool-Framework für binäre Datentypen 41
Stefan Bohne

Functional Kleene Closures 49
Nikita Danilenko

PAF: A portable assembly language based on Forth 51
M. Anton Ertl

Infinity - Eine erweiterbare Programmiersprache 61
Marcus Frenkel

An Efficient Scalable Runtime System for S-Net Dataflow Component Coordination 71
Clemens Grelck

An Efficient Native Function Interface for Java 83
Matthias Grimmer

Control Flow Unfolding of Workflow Graphs Using Predicate Analysis and SMT Solving 93
Thomas S. Heinze

Specification of Real-Time Systems with Mixed Time-Criticality 101
Raimund Kirner

Turning Conjectures into Positive Knowledge: Proving Precision of Worst-Case Execution-Time Bounds 111
Jens Knoop

Extending a Model-Driven Approach for the Cross-Platform Development of Business Apps 113
Herbert Kuchen

A surprising link between functional programming and topology 115
Gunther Schmidt

Finding Security Bugs in Java Programs using Datalog 125
Bernhard Scholz

Implementierung eines Typinferenzalgorithmus für Java 8 127
Andreas Stadelmeier

Automated verification of a relation-algebraic matching program 135
Insa Stucke

Generierung von Informationssystemen mit Integritätsbedingungen und Sicherheitseigenschaften 137
Mathias Weber

Typprüfung mittels Attributgrammatiken auf natürlichsprachlichen Anforderungs-Beschreibungen technischer Systeme 141
Sebastian Wendt

Syntaxanalyse auf Wiedervorlage 151
Baltasar Trancón y Widemann

Abstract / Zusammenfassung

Abstract

This proceedings volume contains the work presented at the 17th Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS'13) at LEUCOREA, Lutherstadt Wittenberg, Germany, from September 29th to October 2nd, 2013.

Zusammenfassung

Dieser technische Bericht fasst die Beiträge des 17. Kolloquiums Programmiersprachen und Grundlagen der Programmierung (KPS'13) zusammen, das vom 29. September bis 02. Oktober 2013 in der LEUCOREA in der Lutherstadt Wittenberg in Deutschland stattgefunden hat.

Vorwort

Dieser technische Bericht fasst die Beiträge des Kolloquiums Programmiersprachen und Grundlagen der Programmierung (KPS'13) zusammen. Das Kolloquium fand letztes Jahr bereits zum 17. Mal statt und setzt eine Reihe von Arbeitstagungen fort, die von den Professoren Friedrich L. Bauer (TU München), Klaus Indermark (RWTH Aachen) und Hans Langmaack (CAU Kiel), der in diesem Jahr seinen 80. Geburtstag feiert, ins Leben gerufen wurde.

Die lange Tradition wird sichtbar in der Liste der bisherigen Veranstaltungsorte:

1980	Tannenfelde im Aukrug	Universität Kiel
1982	Altenahr	RWTH Aachen
1985	Passau	Universität Passau
1987	Midlum auf För	Universität Kiel
1989	Hirschegg	Universität Augsburg
1991	Rothenberge bei Steinfurth	Universität Münster
1993	Garmisch-Partenkirchen	Universität der Bundeswehr München
1995	Alt-Reichenau	Universität Passau
1997	Avendorf auf Fehmarn	Universität Kiel
1999	Kirchhunden-Heinsberg	FernUniversität in Hagen
2001	Rurberg in der Eifel	RWTH Aachen
2004	Freiburg-Munzingen	Universität Freiburg
2005	Fischbachau	LMU München
2007	Timmendorfer Strand	Universität Lübeck
2009	Maria Tafel	TU Wien
2011	Schloss Raesfeld	WWU Münster

Das 17. Kolloquium Programmiersprachen und Grundlagen der Programmierung wurde von der Arbeitrgruppe Software-Engineering und Programmiersprachen des Instituts für Informatik der Martin-Luther-Universität Halle-Wittenberg organisiert.
Tagungsort war die LEUCOREA in der Lutherstadt Wittenberg.
Wir freuten uns, dass wir 30 Teilnehmer begrüßen durften, darunter Prof. Dr. Dr. h.c. Hans Langmaack, einen der Gründungsväter dieser Veranstaltungsreihe. In 21 thematisch vielfätigen Vorträgen und einer Live-Demonstration zur Vorstellung der Programmiersprache MOSTflexPL (Christian Heinlein, HS Aalen) wurde fast das gesamte Spektrum der Programiersprachenforschung abgedeckt.
Die Länge der Beiträge in diesem Tagungsband richtet sich nach den Wünschen der jeweiligen Autoren. Es finden sich sowohl ausfürliche Artikel als auch erweiterte Zusammenfassungen und in einigen wenigen Fällen ist nur eine Zusammenfassung angegeben.

Wir möchten an dieser Stelle ganz herzlich den MitarbeiterInnen der LEUCOREA und dem Betreiber der Cafeteria danken. Räumlich und kulinarisch sind in der Stiftung keine Wünsche offen geblieben.
Danken möchten wir Frau Ramona Vahrenhold, der Sekretärin unserer Arbeitsgruppe, die die Tagung vor- und nachbereitete.

Halle, im April 2014 — Roswitha Picht und Wolf Zimmermann

pylibjit: A JIT Compiler Library for Python

Gergö Barany
Institute of Computer Languages
Vienna University of Technology
gergo@complang.tuwien.ac.at

Abstract

We present pylibjit, a Python library for generating machine code at load time or run time. The library has two distinct modes of use: First, it aims to provide a high-level interface for generating code at run time. This is achieved by using language features such as operator overloading and Python's context managers and decorators.

Second, Python's reflection features allow us to access functions' abstract syntax trees. pylibjit can make use of this to generate machine code for functions originally written in Python. Compiling a Python function can be as easy as attributing it with a decorator providing type information, without changing the function itself in any way. Such compiled functions are executed transparently within interpreted Python programs. This makes it convenient to develop applications in Python and then speed up only the hot spots using compilation.

The development of pylibjit is at a very early stage, but we can already present some working examples. For simple numeric programs, we achieve speedups of up to $50\times$ over standard interpreted Python.

1 Motivation

Interpreted high-level programming languages are often criticized for their comparatively poor performance. Still, they are popular due to their ease of use and high-level features. Various techniques are used to bridge the performance gap to more traditional compiled languages: tracing just-in-time compilers (JITs) [CSR+09, BCFR09], JITs for subsets of languages marked by special annotations [asm], or ahead-of-time compilers that rely on static type annotations [Sel09]. A partial solution is to use interpreters that specialize the program at run time [Bru10].

This paper introduces the pylibjit library, which aims to provide a JIT for fragments of Python code marked by annotations ('decorators' in Python parlance). The compiled code runs within the normal Python interpreter embedded in a traditional Python program; program parts without compiler

```
def fib(n):
    if n < 2:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

Figure 1: The naïve Fibonacci function in Python.

annotations are interpreted as usual, while the compiled parts can make use of static type information provided by the developer.

JITs are mostly associated with languages that are compiled to some form of 'bytecode' intermediate representation. However, machine code generation at run time is an important topic even for some programs written in traditional, statically compiled languages. Code generation can be useful for programs that perform expensive computations with control flow that heavily depends on user input. For example, image processing applications can benefit from generating specialized JIT-ed code for user-defined image filters [Pet07]. Even the Linux kernel contains a JIT compiler for a simple domain-specific language describing network packet filtering rules [Cor11].

Our `pylibjit` project builds on a JIT library meant for building code at run time in such applications. However, since Python also allows execution of code at load time (or rather, since there is no clear distinction between 'run time' and 'load time'), we can apply the same library to compile entire Python function definitions as they appear in a source file. The following sections describe how `pylibjit` builds up from a simple JIT library interface to what is a de facto ahead-of-time compiler for a subset of Python.

2 Compiling functions using the low-level API

Our `pylibjit` library builds on the GNU `libjit` just-in-time compiler library[1] and an existing Python wrapper library for it[2].

We will use the naïve Fibonacci function in Figure 1 to illustrate the use of this existing library and our improvements to it. The library exports a class `jit.Function` which users must subclass for each function they wish to build and compile. Figure 2 shows the code needed to build the Fibonacci function using this interface. The core is the `build` function which performs the API calls to build up the intermediate code step by step: API calls provide for computations, labels, conditional and unconditional jumps, and function calls. Note that, since the intermediate code is meant to be compiled to machine code, all computations are typed, although type annotations are

[1] `http://www.gnu.org/software/libjit/`

[2] `https://github.com/eponymous/libjit-python`

```
class fib_function(jit.Function):
    def create_signature(self):
        # return type, argument types
        return self.signature_helper(jit.Type.int, jit.Type.int)

    def build(self):
        # values: n, one, two
        n = self.get_param(0)
        one = self.new_constant(1, jit.Type.int)
        two = self.new_constant(2, jit.Type.int)
        # if n < 2: goto return_label
        return_label = self.new_label()
        self.insn_branch_if(n < two, return_label)
        # return fib(n-one) + fib(n-2)
        fib_func = self
        fib_sig = self.create_signature()
        a = self.insn_call('fib', fib_func, fib_sig, [n-one])
        b = self.insn_call('fib', fib_func, fib_sig, [n-two])
        self.insn_return(a + b)
        # return_label: return n
        self.insn_label(return_label)
        self.insn_return(n)
```

Figure 2: Building the Fibonacci function using the low-level JIT interface.

mostly only needed where a value is first introduced; the types of arithmetic and other operators are inferred from their operands. The type name `int` refers to the machine's 'usual' word-sized integer type, as in C.

Note also that already at this level, Python's high-level nature offers some convenience: The arithmetic operators `+` and `-` (and others) are overloaded to work on the compiler's 'value' objects, so we can directly generate an addition instruction by writing `a + b` rather than the less natural `insn_add(a, b)`.

Having created this definition, the class can be instantiated and called as if it were a normal Python function. Wrapper code takes care of unboxing Python argument values and boxing the native code's return value in a Python object.

3 The higher-level API

While this library is usable, it makes building functions more verbose than absolutely necessary. Having to subclass `Function` and implement several methods on it can lead to a lot of boilerplate code; the only interesting function in a typical subclass is `build`, so ideally users should not have to write more than this function. Conveniently, Python provides a concept of *function decorators* that can be used for this purpose.

```
@jit.builder(return_type=jit.Type.int, argument_types=[jit.Type.int])
def fib2(func):
    n = func.get_param(0)
    one = func.new_constant(1, jit.Type.int)
    two = func.new_constant(2, jit.Type.int)
    with func.branch(n < two) as (false_label, end_label):
        func.insn_return(n)
    # else:
        func.insn_label(false_label)
        func.insn_return(func.recursive_call('fib', [n - one]) +
                         func.recursive_call('fib', [n - two]))
```

Figure 3: Building the Fibonacci function using the higher-level interface.

A decorator is a callable object that can be attached to a function definition using the `@` operator. After the function has been parsed and compiled to Python bytecode, the decorator is applied to it and can perform any analysis or transformation. Finally, the decorator's return value replaces the original function object. This enables various higher-order programming techniques.

Using this mechanism, it is easy to write a decorator for JIT builder functions that hides all the boilerplate involved in defining a class and instantiating it. This is encapsulated in the `jit.builder` decorator exported by `pylibjit`. This decorator creates an internal class subclassing `jit.Function`, defines the `build` method in that class to call the decorated function provided by the user, and finally takes care of instantiating the JIT-ed function.

Besides boilerplate, another inconvenience when using the pure `libjit` interface is the lack of structure. In Figure 2, the control flow in the generated program is difficult to see as it is implicit in a number of labels and jump statements. We can, however, build constructs for more structured programs using Python's *context managers* combined with its `with` statement. In essence, a context manager is a pair of two functions `__entry__` and `__exit__` which are called when execution enters and leaves a `with` statement, respectively.

This is a good match to the work that must be done to set up a branch or a loop: At the head of the control structure, a branch condition must be evaluated, and at the end a label (and, for loops, a jump back to the loop head) must be generated. `pylibjit` defines context managers `branch` and `loop` for this purpose. Figure 3 shows how the code building the Fibonacci function can be simplified by using a decorator and the `branch` context manager. The context manager's entry function generates the labels needed to distinguish the true and false branches of execution; unfortunately, the abstraction is not perfect because the user must still take care of placing some jumps and labels.

This interface makes it convenient to build functions at run time and, if needed, specialize them to user input that is partly known (such as user-defined filters [Pet07]). However, the API is still too verbose for functions encapsulating algorithms that we do not want to specialize in this way. The next section puts everything together by showing how `pylibjit` can leverage Python's own syntax for specifying compiled functions.

4 Compiling Python functions

As the final step in the development, we note that Python's `inspect` module allows function decorators to access metadata for functions. In particular, it can be used to obtain the function's original source code (if the program is available in source form, which is typically the case); that code can be extracted and passed to the `ast` module to obtain an abstract syntax tree (AST) for the function [BJ13].

We are therefore in a position to write a decorator function which accesses its decorated function's AST and traverses that AST emitting appropriate `pylibjit` instructions for each AST node. That is, we obtain a very simple way to replace interpreted Python functions by compiled code implementing the same semantics. The Python code itself need not change at all: Whether the code is interpreted or compiled depends only on whether an appropriate decorator is present.

A compiled version of the Fibonacci function is shown in Figure 4. Note, again, that apart from the decorator it is identical to the original Python implementation in Figure 1. At any point during development, the decorator can be removed (e. g., by commenting it out) to switch back to the original interpreted function, or inserted again to obtain the benefits of compilation.

As with all the previous examples, this also produces an object that can be called like a function. On our development machine (Intel Atom N270 at 1.60 GHz running Linux 3.2.0), it takes about 8.2 seconds to evaluate `fib(32)` for the original Python version, while the compiled version takes about 0.155 seconds, for a speedup of about 53 ×.

```
@jit.compile(return_type=jit.Type.int, argument_types=[jit.Type.int])
def fib(n):
    if n < 2:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

Figure 4: The compiled naïve Fibonacci function.

```
def eval_A (i, j):
    return 1.0 / (((i + j) * (i + j + 1) >> 1) + i + 1)

def eval_A_times_u (u, results):
    u_len = len (u)
    for i in range (u_len):
        partial_sum = 0
        j = 0
        while j < u_len:
            partial_sum += eval_A(i, j) * u[j]
            j += 1
        results[i] = partial_sum
```

Figure 5: The `spectral_norm` benchmark.

5 Larger benchmarks

The ultimate goal for `pylibjit` is to be able to compile a nontrivial subset of Python to speed up the innermost loops of computationally intensive programs. While development is at a very early stage, we can already use it to speed up some interesting benchmark applications. The decorator-based approach with an embedding in the Python interpreter is convenient for incremental development of the compiler: We never need to worry about handling all the intricacies of the Python language, only the features actually used in the few functions of interest.

At the time of writing, we can compile Python functions containing arithmetic, branching, counting loops, function calls, and accesses to tuples, lists, and arrays. Compilation is implemented using an AST traversal comprising about 600 lines of code. Here we present results for two benchmark programs in which the hot spots use only these features.

5.1 Case study: The `spectral_norm` benchmark

First, consider the implementation of the core of the `spectral_norm` benchmark[3] in Figure 5. The `eval_A_times_u` function takes two arrays of floating-point numbers as its arguments and populates the second based on values from the first. The other half of the benchmark is an almost identical function that differs only in that the arguments `i` and `j` to `eval_A` are swapped.

Running this benchmark with an input of 1000 (denoting the length of the input array) takes 190.7 seconds using the Python interpreter. We can cause the core functions to be compiled by `pylibjit` by defining a decorator

[3] `http://benchmarksgame.alioth.debian.org/u32/program.php?test=spectralnorm&lang=python3&id=8`

that declares the types of the arguments and variables used within each function.

Decorating only the `eval_A` function, the benchmark's performance deteriorates to 304.9 seconds, a $1.6\times$ slowdown. This is caused by the fact that calls from Python to code compiled by pylibjit are expensive. Such calls follow a slow path in the interpreter in which the code to be called is looked up in an object's `__call__` slot. The wrapper code must then check and unbox the Python argument objects and finally box up the function's result in a Python object. It is therefore not worthwhile to compile only very small leaf procedures such as `eval_A`.

However, if we also compile the outer `eval_A_times_u` function and its almost identical twin, we obtain a version of the program that runs in 3.5 seconds. This is a speedup of about $53\times$ over the Python interpreter. The time for the compiled version includes JIT compilation time, which for this benchmark is on the order of 0.05–0.1 seconds. For comparison, the C version of this benchmark compiled with GCC 4.6.3 takes about 3.4 seconds when compiled at `-O1`, 2.8 seconds at `-O2`, and 2.6 seconds at `-O3`. This shows that for numerical computations on array-like structures we can sometimes come close to the performance of optimized C code while enjoying the benefits of developing in Python.

5.2 Case study: The `AES` benchmark

As a larger and more realistic application, we compiled parts of a Python implementation of the AES encryption algorithm[4]. AES encryption and decryption consists essentially of XOR operations and permutations of arrays of bytes. Figure 6 shows two representative functions. Compiling these and a few other similar leaf procedures which together account for 81 % of execution time, `pylibjit` improves benchmark runs from 12.4 seconds (interpreted) to 3.42 seconds for a speedup of about $3.6\times$. (In other words, 72 % of execution time is optimized away.)

This can be improved further by also compiling the outer loops that call such leaf functions. In this implementation of AES, the entire algorithm is encapsulated in a Python class, and the calls to the auxiliary functions are therefore implemented as virtual method calls. However, in the context of this benchmark, the targets for these calls can, in principle, always be determined statically. `pylibjit` allows users to annotate such functions to resolve targets at compile time. This is useful because it saves not only lookup time but also surprisingly expensive boxing and unboxing operations on method objects [Bar13].

Applying this user-guided static devirtualization when compiling the encryption and decryption driver functions, we obtain a total benchmark ex-

[4] Adapted to Python 3 from a version available from `https://bitbucket.org/pypy/benchmarks/src`

```
def add_round_key(self, block, round):
    offset = round * 16
    exkey = self.exkey
    for i in range(16):
        block[i] ^= exkey[offset + i]

def sub_bytes(self, block, sbox):
    for i in range(16):
        block[i] = sbox[block[i]]
```

Figure 6: Two of the hottest functions in the `AES` benchmark, showing simple manipulation of arrays of bytes.

ecution time of only 0.607 seconds. This corresponds to a total speedup of about $20\times$ versus interpretation.

6 Conclusions and future work

We presented `pylibjit`, a Python wrapper for the GNU `libjit` just-in-time compiler library. Besides just exposing the underlying API, `pylibjit` allows the use of decorators to cause existing Python functions to be compiled to machine code. At the time of writing, the library supports a fragment of Python supporting integer and floating-point arithmetic, counting loops, function calls, and accesses to Python tuples, lists, and arrays.

The current version of `pylibjit` is not fit for general use, but the compiler is simple and extensible and will soon grow more features as needed. Since the compiled code runs within the Python interpreter, all of the functions used internally by Python to implement operations are accessible and can be called from the compiled code. This means that to support more language features, `pylibjit` need only mimic the sequence of internal API calls that the interpreter would itself execute for a given program. However, the presence of type annotations means that we can elide certain operations that add overhead, such as dynamic typechecks and boxing/unboxing operations.

A more complete version of `pylibjit` will be made available through the author's website at `http://www.complang.tuwien.ac.at/gergo/`.

Acknowledgements

This work was supported by the Austrian Science Fund (FWF) under contract no. P23303, Spyculative.

References

[asm] asm.js: an extraordinarily optimizable, low-level subset of JavaScript. `http://asmjs.org/`.

[Bar13] Gergö Barany. Static and dynamic method unboxing for Python. In *6. Arbeitstagung Programmiersprachen (ATPS 2013)*, included in volume 215 of *Lecture Notes in Informatics (LNI)*. GI - Gesellschaft für Informatik, February 2013.

[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*, ICOOOLPS '09, pages 18–25, New York, NY, USA, 2009. ACM.

[BJ13] David Beazley and Brian K. Jones. *Python Cookbook, 3rd Edition*, chapter Parsing and Analyzing Python Source. O'Reilly, 2013.

[Bru10] Stefan Brunthaler. Efficient interpretation using quickening. In *Proceedings of the 6th symposium on Dynamic languages*, DLS '10, pages 1–14, New York, NY, USA, 2010. ACM.

[Cor11] Jonathan Corbet. A JIT for packet filters. `https://lwn.net/Articles/437981/`, 2011.

[CSR+09] Mason Chang, Edwin Smith, Rick Reitmaier, Michael Bebenita, Andreas Gal, Christian Wimmer, Brendan Eich, and Michael Franz. Tracing for web 3.0: trace compilation for the next generation web applications. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 71–80, New York, NY, USA, 2009. ACM.

[Pet07] Charles Petzold. On-the-fly code generation for image processing. In Andy Oram and Greg Wilson, editors, *Beautiful Code*. O'Reilly, 2007.

[Sel09] Dag Sverre Seljebotn. Fast numerical computations with Cython. In *Proceedings of the 8th Python in Science conference (SciPy 2009)*, 2009.

Konsistenzprüfung bei der Anforderungsfindung für hydromechatronische Systeme

Christian Berg Wolf Zimmermann

Martin-Luther-Universität Halle-Wittenberg
Institut für Informatik
Lehrstuhl für Software-Engineering und Programmiersprachen

Zusammenfassung

Die Entwicklung hydromechatronischer Systeme, die wir in dieser Arbeit beschreiben, erfolgt iterativ und hat daher an vielen Stellen mögliche Punkte für daraus resultierende Inkonsistenzen. Mit Hilfe von Tiefensuche und Übersetzerbautechnologie können wir in der ersten Phase dieser Iterationen mögliche Inkonsistenzen frühzeitig finden. Wir vergleichen unsere Implementierung auf Basis der Methoden des Übersetzerbaus mit den Methoden die im Modell-basierten Umfeld genutzt werden[10]. Wir zeigen, dass unsere Umsetzung vom Aspekt der Geschwindigkeit und Mächtigkeit für die Größenordnungen im hydromechatronischen Umfeld geeigneter sind.

1 Einleitung

Wie in anderen Industrien werden hydromechatronische Systeme inkrementell entwickelt. Im Gegensatz zum Automobilbau, sind die Anforderungen an ein hydromechatronisches Produkt jedoch andere: wie im Automobilbau sind die gesetzlichen Grenzen verglichen mit der Flugzeugindustrie vergleichsweise gering, jedoch die Garantiezeiten wesentlich höher (zehn Jahre für das Gesamtprodukt). Im Gegensatz zum Automobilbau, der schon seit 1976 Software verwendet[20], ist die Verwendung von Software in einem hydromechanischen Produkt ein sehr junges Thema. Die inhärente Komplexität hochintegrierter Systeme sorgt für viele Stellen an denen Inkonsistenzen im Entwicklungsprozess vorkommen können[19]. Einige dieser Inkonsistenzen bestehen darin herauszufinden ob eine Anforderung erfüllt oder nicht erfüllbar ist. Konsistenzprüfung für den gesamten hydromechatronischen Produktentstehungsprozess ist ein aktuelles Forschungsthema, daher behandeln wir in dieser Arbeit einen Ausschnitt dieses Entwicklungsprozesses und zeigen wie mit Methoden des Übersetzerbaus und statischen Analysen mögliche Inkonsistenzen gefunden werden können. Wir zeigen, dass die von uns genutzten Methoden des Übersetzerbaus zur Analyse geeigneter sind als entsprechende Standardmethoden aus dem Modell-basierten Umfeld.

Unser Beitrag in dieser Arbeit ist

- die Vorstellung des hydromechatronischen Produktentstehungsprozess im Kontext der Informatik und dabei bestehende Probleme (Abschnitt 2)
- eine Abbildung dieses (Teil-)Prozesses mit Methoden des Übersetzerbaus in Abschnitt 3,
- das exemplarische Zeigen zweier Möglichkeiten der Verwendung von Methoden des Übersetzerbaus zum Finden von Inkonsistenzen (Abschnitt 4)
- und, dass wir zeigen, dass auch für große Beschreibungen die Methoden des Übersetzerbaus den Methoden aus dem Modell-basierten Umfeld überlegen sind (Abschnitt 5).

Wenngleich unser Ansatz nicht gänzlich neu ist, so zeigen wir doch in Abschnitt 6, dass momentan viele der Möglichkeiten des Übersetzerbaus in dem Umfeld der Mechatronik nur unzureichend genutzt werden. Letztendlich geben wir in Abschnitt 7 eine kurze Zusammenfassung und gehen auf weitere aktuelle Fragestellungen ein.

2 Hydromechatronische Produktentstehung

Soweit nicht anders angegeben sind die hier vorgestellten Informationen aus [17] wiedergegeben oder spiegeln eigene Erfahrungen wider, sind jedoch auf die für diese Arbeit notwendigen Informationen verkürzt.

Jede Produktentwicklung startet mit Ideen und Vorschlägen, welche in einer Liste von Anforderungsschlagworten (Schlagworte, die an die eigentliche Anforderung erinnern sollen) gesammelt werden. Wird die Entscheidung getroffen ein Produkt umzusetzen, werden auf Basis dieser Schlagworte alle anderen Dokumente und Artefakte zur Pumpenherstellung (Dokumentation, Gehäuse, Software usw.) erzeugt.

Der Prozess selbst lässt sich wie folgt umschreiben: durch Kommunikation und Verfeinerung wird aus einem Anforderungsschlagwort eine Anforderung spezifiziert, diese bekommt Planungsinformationen (bspw. Kosten, Zeit) sowie Testfälle; die Umsetzung einer Anforderung wird von dem zugewiesenen Personal durchgeführt und in Abstimmung mit der Testabteilung und den Anforderungsverantwortlichen geprüft.

Einen Ausschnitt aus einer künstlichen Anforderungsliste zeigt Abbildung 1.

guid	label	desc	status	priority	refs	bug	author	owner
g-2036723077	nice poster	we want to present our project with a nice poster	new	0	layout,pictures,printing,text		Wolf Zimmermann	
g-1091832359	layout				pictures,text			
g-2020098232	pictures				layout			

Abbildung 1: Ausschnitt aus einer künstlichen Anforderungsliste

Die Notwendigkeit guter Anforderungen und Anforderungsspezifikationen ist allgemein [19] und auch bei der hydromechatronischen Produktentstehung bekannt[17].

Allein aufgrund der großen Anzahl an Anforderungen ist es für Menschen nur begrenzt ohne maschinelle Hilfe möglich, zu prüfen, ob alle ca. 2000 Anforderungen eines aktuell entwickelten Produkts umgesetzt werden. Bei unserem Projektpartner haben hydromechatronische Produkte zwischen 500 und 2500 Anforderungen. Bereits existierenden Werkzeugen und Methoden fehlt ein ganzheitliches Modell. Die genutzte Kommunikation allein reicht bei dem inkrementellen Prozess nicht aus, um sicherzustellen, dass Anforderungen überhaupt umsetzbar sind.

3 Prozessabbildung

Im Rahmen dieser Arbeit bestehen Anforderungen aus der informellen Beschreibung der Anforderungsspezifikation und Daten für das Projektmanagement.

Mit Hilfe des Compilerbau-Werkzeugkastens eli[1] [9] haben wir für eine Sprache zur Abbildung der Anforderungsdaten einen Übersetzer entwickelt. Der Übersetzer dient vornehmlich der Analyse dieser Daten, erzeugt allerdings auch graphische Ausgaben und eine alternative tabellarische Ansicht der Daten.

Diese alternative Ansicht wird genutzt um weniger versierte Anwender an diese Art formaler Sprache heranzuführen. Es existiert für diese Darstellung ebenfalls ein Übersetzer der diese wieder zur textuellen Darstellung überführt. Die textuelle Darstellung der Sprache ist an die Javascript Object Notation (JSON) angelehnt, sodass einfache Weboberflächen schnell erstellbar sind.

Quelltext 1 zeigt den wesentlichen Ausschnitt aus der Grammatik zur Abbildung der Anforderungsliste.

```
⟨ReqTag⟩            ::= 'rq' label ⟨Optuid⟩
                      | 'rq' ⟨Optlabel⟩ ⟨Optguid⟩ '{' ⟨ReqStats⟩ '}'
⟨Optlabel⟩          ::= label | ⟨empty⟩
⟨Optuid⟩            ::= uid | ⟨empty⟩
⟨ReqStats⟩          ::= ⟨ReqStatement⟩ | ⟨ReqStatement⟩ ⟨ReqStats⟩
⟨ReqStatement⟩      ::= 'desc' ⟨Optdot⟩ text
                      | 'label' ⟨Optdot⟩ label
                      | 'uid' ⟨Optdot⟩ guid
                      | 'bug' ⟨Optdot⟩ integer
                      | 'ref' ⟨Optdot⟩ (⟨Reference⟩ | ⟨References⟩)
⟨Reference⟩         ::= guid | label
⟨References⟩        ::= '[' ⟨ReferenceList⟩ ']'
⟨ReferenceList⟩     ::= ⟨Reference⟩
                      | ⟨Reference⟩ ⟨ReferenceList⟩
```

Quelltext 1: Ausschnitt der Grammatik zur Abbildung von Anforderungsschlagworten

[1] http://eli-project.sourceforge.net

Mit Hilfe geordneter attributierter Grammatiken[2][15] werden die Anforderungen analysiert und verschiedene Ausgabeformate (graphisch in Form von DOT[3], tabellarisch in Form von CSV[4]) erzeugt.

```
rq "nice poster" {
  desc : "we want to present our project with a nice poster"
  ref : ["layout" "pictures" "printing" "text"]
  aut : "Wolf Zimmermann"
  state : new
  pri : 0
}
rq "layout" {ref : ["pictures" "text"]}
rq "pictures" {ref : "layout" }
rq "printing"
rq "text" {
  desc : "we need to present a readable text"
  own : "Christian Berg"
  bug : 12313
}

rt rq "nice poster"
```

Quelltext 2: Ausschnitt einer Beschreibung in der entwickelten Sprache

Zur Analyse notwendig ist die Einführung einer „Oberanforderung“, die, sollte sie noch nicht existieren, durch die Anforderung „Erzeuge Druck und Durchfluss“, oder ähnliche, realisiert werden kann. Dies wird gekennzeichnet durch das Schlüsselwort `rt`.

Ein Beispiel einer Beschreibung zeigt Quelltext 2. Die aus dem Beispiel erzeugte graphisch Repräsentation findet sich in folgender Abbildung (Abb. 2). In folgenden Versionen beabsichtigen wir die Hinweismarkierungen ebenfalls automatisch hinzuzufügen.

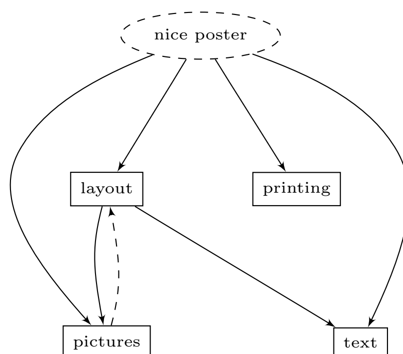


Abbildung 2: Angepasste Version der automatisch generierten Ausgabe für Quelltext 2 (Kantenstil zu gestrichelt geändert)

[2] engl. ordered attribute grammar (OAG)
[3] DOT ist eine Beschreibungssprache für Graphen, siehe auch http://www.graphviz.org
[4] engl. comma-separated values

4 Statische Analysen zur Konsistenzprüfung

Da ein hydromechatronisches Produkt inkrementell entwickelt wird, entwickeln sich auch Anforderungen und ebenso Anforderungsschlagworte fortlaufend weiter. Üblicherweise gibt es eine Hierarchie von Anforderungen, sodass eine Anforderung von anderen Anforderungen abhängig ist. Im Verlauf des Projekts ist dies ein Punkt durch den Inkonsistenzen entstehen können: wird eine bestehende Anforderung entfernt sind Anforderungen, die von dieser abhängig sind nicht mehr erfüllbar. Fügt man jedoch neue Anforderungen und Abhängigkeiten hinzu ist es leicht dafür zu sorgen, dass eine Anforderung sich transitiv selbst als Abhängigkeit enthält, sodass diese gleichfalls nicht erfüllbar ist. Im Folgenden werden die Analysen vorgestellt, die sicherstellen, dass Anforderungen, die ursächlich für diese Probleme sind gefunden werden können.

Soll die natürlichsprachliche Beschreibung der Anforderungsspezifikation beibehalten werden und keine Verarbeitung natürlichsprachlichen Textes[5] erfolgen, gibt es drei Ansatzpunkte um Stellen für Inkonsistenz zu finden:

- unbekannte Anforderungen,
- toter Anforderungen,
- sowie zyklische Anforderungen.

Unsere Arbeit erfolgt unabhängig von der natürlichsprachlichen Beschreibung, sodass nur auf der (Abhängigkeits-)Struktur – als Graph – geprüft wird.

Im Folgenden werden die Konsistenzkritieren formal anhand der graphischen Repräsentation dargelegt.

Definition 1. Ein **Anforderungsgraph** $G(V, W, E, r)$ ist ein gerichteter Graph $G(V, E)$ mit V Knoten, $W \subseteq V$ explizit beschriebene Knoten, $E \subseteq V \times V$ Kanten und einem augezeichneten Knoten $r \in W$. Die Knoten $V \subseteq \mathit{Label} \times \mathit{Guid}$ bilden die Beschreibung der Anforderungsschlagworte ab. Eine Anforderung rq ist in V genau dann, wenn sie in dem Modell des hydromechatronischen Produkts explizit beschrieben ist oder von einer explizit beschriebenen Anforderung via `ref` als Abhängigkeit markiert ist. Eine Anforderung rq ist ein W genau dann, wenn sie in dem Modell explizit beschrieben ist (eingeleitet durch das Schlüsselwort **rq**).

Definition 2. **Erstes Konsistenzkriterium** $V = W$. Die Mengen V und W sind identisch.

Das erste Konsistenzkriterium sagt also aus, dass es kein Anforderungsschlagwort gibt, welches nicht explizit beschrieben ist.

Die Folgende Eigenschaft und der dazugehörige Relationsbegriff werden in den folgenden Konsistenzdefinitionen benötigt.

Definition 3. **Abschlussrelation des Anforderungsgraphs**: Sei $R \subseteq V \times V$ die durch E definierte Relation, d.h. $\forall u, v \in V$ mit $(u, v) \in E$, dann gilt uRv. Seien die Anforderungen $a, b, c \in V$ gegeben, dann $aRb \wedge bRc \Rightarrow aRc$.

Definition 4. **Zweites Konsistenzkriterium** Sei $v_0 = r$, dann gilt $\forall v_i \in V \setminus \{v_0\} : v_0 R v_i$.

[5] engl. natural language processing (NLP)

Das zweite Konsistenzkriterium lässt sich graphentheoretisch wie folgt ausdrücken: Sei $v_0 = r$, dann gilt $\forall v_t \in V \setminus \{v_0\}$ mit $W_t = (v_0, ..., v_t)$ ist ein Weg, sodass für alle $i \in \{1, ..., t-1\}$ gilt $E((v_i, v_i + 1)) > 0$.

Das letzte Konsistenzkriterium lässt sich vereinfacht dadurch ausdrücken, dass keine Anforderung sich selbst im transitiven Abschluss über die via `ref` markierten Anforderungen enthält:

Definition 5. Drittes Konsistenzkriterium $\forall r_a, r_b \in V : r_a R r_b \Rightarrow r_a \neq r_b$.

Das erste Konsistenzkriterium (Definition 2) lässt sich mit einfacher Namensanalyse auf den Schlagworten und eindeutigen IDs überprüfen. Hierfür gibt es bereits vorgefertigte Module [16]. Die von uns verwendete Namensanalyse erlaubt Nutzung vor Definition.

Die Prüfung ob eine Anforderung eingehende Kanten hat, bspw. ausschließlich über Seiteneffekte auf der Definitionstabelle (zählen der eingehenden Kanten eines Anforderungsschlagwortes), ist allein nicht ausreichend: wird das dritte Konsistenzkriterium verletzt ist es leicht Beispiele zu konstruieren, bei denen alle Anforderungen eingehende Kanten haben aber das zweite Konsistenzkriterium verletzt wird. Eine Anforderung, die nicht referenziert wird, invalidiert die eigenen Referenzen.

Da Konsistenzkriterium zwei und drei daher eng miteinander verknüpft sind werden diese geprüft indem bei den Attributberechnungen als Seiteneffekte den eigentlichen Graph ohne weitere Informationen aufbaut. Dieser Graph wird dann, wie der abstrakte Syntaxbaum selbst, mit Tiefensuche(siehe u.a. [4]) durchlaufen. Durch Verwendung des Visitor-Patterns[14] wird bei dem Vorhandensein von Rückkanten – das dritte Konsistenzkriterium wird nicht eingehalten – die Fehlerausgabe um den kompletten Zyklus angereichert. Tiefensuche und das Visitor-Pattern liegen auch attributierten Grammatiken zugrunde[25].

Es ist möglich die Operationen, die von uns mit eli durch Seiteneffekte durchgeführt wurden, mit Hilfe von Baumtransformationen und reinen Attributberechnungen durchzuführen, jedoch fehlt dafür in eli (bisher) die Unterstützung.

5 Evaluation

Wir vergleichen die Geschwindigkeit der Analysen aus dem Übersetzerbau und Graphentheorie mit den Möglichkeiten, die im Eclipseumfeld ohne Codeanpassung möglich sind. Das Testsystem ist ein Sabayon Linux mit Kernel Version 3.9. Das System hat eine Intel Core i7-3770 CPU und verfügt über 16 GiB Arbeitsspeicher.

Mit Hilfe eines Testgenerators wurden verschiedene Dateien erstellt. Parametrierbar ist in diesem Testgenerator die Anzahl der zu generierenden Anforderungsschlagworte, die Anzahl der Verbindungen und die Anzahl toter Anforderungen in Prozent sowie ob Zyklen erlaubt sind oder nicht. Eine Dichte von 100% entspricht einem vollständigen Graphen, 100% toter Knoten bedeutet, dass es keine Kanten gibt. Der Testgenerator generiert anhand der übergebenen Parameter und zufälligen Werten die konkrete Ausprägung der Anforderungsbeschreibung.

Für die Modell-basierte Entwicklung wurde ein ecore-Metamodell, abgebildet in Abbildung 3, umgesetzt, was, soweit möglich, der Grammatik aus Abschnitt 3 entspricht. Für die Konsistenzprüfung wurde OCL eingesetzt. Durch den Testgenerator wurden die ecore-Modelle sowie die programmiersprachlichen Beschreibungen mit denselben Daten erzeugt. OCL (OCLinEcore) wurde eingesetzt, da damit der generierte Code nicht angepasst werden muss und OCL auch zur Spezifikation der Semantik von Modellen eingesetzt werden kann[10].

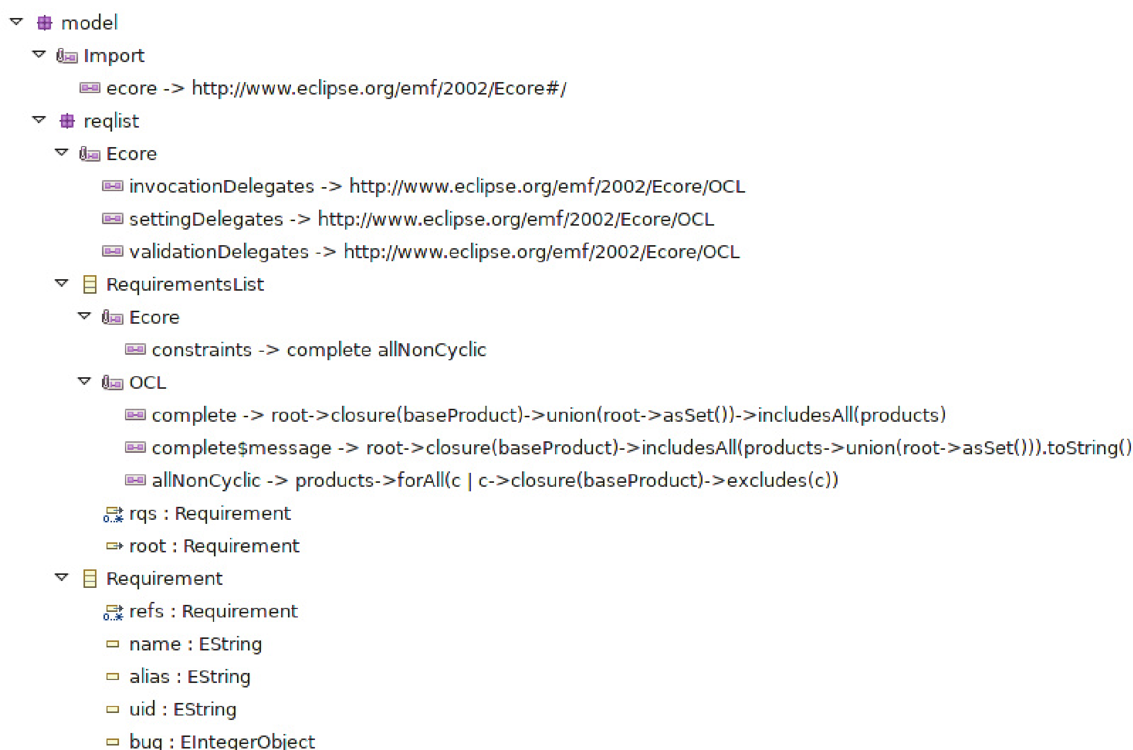


Abbildung 3: ecore Metamodell und OCL-Beschreibungen

Zur Prüfung des ersten und zweiten Konsistenzkriteriums kommt folgender OCL Ausdruck zum Einsatz: `self.root->closure(refs)->union(self.root->asSet())->includesAll(rqs);` Zur Prüfung des dritten Konsistenzkriteriums kommt

```
invariant nonCyclic :
  allRqs->forAll(c |
          c->closure(refs)->excludes(c));
```

zum Einsatz, welches nahezu identisch zu der in [5] vorgestellten Variante zum Sichern der Azyklizität ist[6]. Eine algorithmische Umsetzung der Prüfung des dritten Konsistenzkriteriums wurde in Ecore nicht durchgeführt, weshalb die Werte in Tabelle 1 gesondert aufgeschlüsselt wurden. Die Erzeugung sortierter Ausgaben wurde für die Evaluation abgeschalten, umfangreiche Prüfungen finden in unserer Variante dennoch statt (z.B. das Finden mehrerer Wurzeln, Nutzung nur definierter Anforderungen, usw.).

[6]Die Variante aus [5] schiebt die Prüfung eine Ebene tiefer (in unserem Fall zu den Anforderungen selbst), sodass nicht vorzeitig abgebrochen werden kann – geringere Geschwindigkeit für explizite Aussagen wo die Bedingungen fehlschlagen wäre die Folge.

Knoten	Dichte	Tot	zyklisch	ecore K2	eli K2	ecore gesamt	eli gesamt
100	3	10	nein	1,46 s	< 0,01 s	1,477 s	< 0,01 s
1000	4	2	nein	1,63 s	0,04 s	6,02 s	0,04 s
1000	4	2	ja	1,63 s	0,045 s	1,82 s	0,04 s
2000	6	10	nein	1,78 s	0,23 s	118,61 s	0,22 s
2000	10	0	nein	2,80 s	0,45 s	534,51 s	0,45 s
5000	10	10	nein	2,78 s	2,61 s	*>20 m*	2,59 s
10000	2	4	nein	3,10 s	3,16 s	*>20 m*	3,14 s
10000	2	4	ja	2,94 s	2,94 s	2,96 s	2,75 s

Tabelle 1: Laufzeiten der Werkzeuge zur Konsistenzprüfung

Wie bei OCL brechen wir ebenfalls ab, sobald der erste Zyklus gefunden wurde, jedoch geben wir nicht nur an, dass ein Zyklus gefunden wurde, sondern auch welche Elemente Teil des Zyklus sind. Gleiches gilt für die Prüfung des zweiten Konsistenzkriteriums, hierbei brechen wir ebenfalls ab, sollten Anforderungen „tot“ sein, geben jedoch auch aus, welche Anforderung dies betrifft.

Bis auf die kursiv markierten Werte in Tabelle 1 wurden für die Gesamtlaufzeiten 1000 Testläufe durchgeführt und das arithmetische Mittel der Laufzeit angegeben. Dauerte ein einzelner Test länger als 20 Minuten wurde dieser abgebrochen und neu gestartet, nach zehn solcher Versuche wurde der gesamte Testlauf abgebrochen; in der Tabelle sind diese Testläufe kursiv markiert.

Auffällig sind die Punkte an denen OCL nicht vorzeitig abbrechen kann, bspw. wenn kein Zyklus vorhanden ist und auch keine Knoten, die nicht im Anforderungsgraphen vorkommen: die Laufzeiten sind sehr hoch. Auch scheint die Initialisierung der OCL-Prüfungen (ecore generiert die OCL-Validierung als Laufzeitausdruck, sodass dieser erst noch durch den eigentlichen Validator geprüft werden muss) und das Einlesen der Daten in den kleinen Fällen (< 2000 Elemente mit geringer Dichte) zu überwiegen.

Eine algorithmische Variante mit XText wurde ebenfalls umgesetzt, diese hat jedoch Probleme aufgrund des Editors und der für die algorithmische Umsetzung benötigten Graphdaten – der Speicher reicht nicht aus und die generierte Editoranwendung ist so träge, dass ein Arbeiten damit unmöglich ist, da bei *jeder* noch so kleinen Änderung Wartezeit im Bereich von Minuten für die großen Beispiele vorkommen. Die Trägheit äussert sich bspw. darin, dass bei dem letzten Dokument von Tabelle 1 auch ohne Validierungen das Hinzufügen *einer* neuen Zeile ca. vier Minuten in Anspruch nimmt und das Testsystem auf nahezu voller Prozessorauslastung läuft. Wir können bisher nur mutmaßen was hier ursächlich ist und planen weitere Tests um dies herauszufinden.

Wie bereits in Abschnitt 2 angedeutet, ist die Größe der Beispiele sogar realitätsnah, da zwar eine geringere Dichte des Graphen der Fall sein sollte, jedoch die Anzahl der Elemente (Anforderungsschlagworte können zu wesentlich mehr Anforderungen führen) praxisrelevant ist.

6 Verwandte Arbeiten

Einen Überblick über den eigentlichen Entwicklungsprozess im hydromechatronischen Umfeld gibt [17] (siehe auch Abschnitt 2). Einen Überblick über die Herausforderungen, denen wir uns im aktuellen Projekt stellen müssen gibt [20], wenngleich beim Automobilbau einige Anforderungen anders sind als in der Hydromechatronik[17]. Die Notwendigkeit eines ganzheitlichen Ansatzes ohne Modellbruch wird in [22] vorgestellt – einer der Gründe weshalb wir Übersetzerbau-Technologie nutzen und nicht die aus dem Eclipseumfeld bewährte Methode der „vielen kleinen DSLs verknüpft über Modelltransformationen“.

Die Grundlage für [22] bildet [7] in der Ebert und Franzke die zugrundeliegende Struktur vorstellen. Dieser Graph kann auch als attributierten Syntaxbaum aufgefasst werden. Grundlage unserer Ideen war bisher im wesentlichen Partsch [19]. In [19] erfolgt ebenfalls ein Überblick über die Herausforderungen Modell-basierter Entwurfsmethoden, was Anforderungsfindung und Anforderungsanalyse einschließt.

In [6] werden nochmal die notwendigen Punkte, die Werkzeuge zum Anforderungsmanagement leisten sollen durch Ebert et. al. aufgezählt. Einen Überblick über Möglichkeiten in OCL Konsistenzbedingungen anzugeben gibt Costa in [5]. Dass sich OCL zur Beschreibung der Semantik von Modellen im Modell-basierten Umfeld nutzen lässt, wird in [10] beschrieben. Dass OCL auch praktisch eingesetzt wird um Programmiersprachen zu analysieren führt [18] auf.

In [2] wird auch darauf eingegangen, dass sich Referenzattributgrammatiken[7] nutzen lassen um die Semantik von Modellen zu beschreiben. An dieser Arbeit hervorzuheben ist die Motivation für diese Arbeit, denn *in der Welt der Metamodellierung sind die formalen Ansätze Spezifikation von statischer Semantik noch nicht etabliert.* Zwar wird in [2] beschrieben, dass das zugrundeliegende Werkzeug auch Graphtransformationen unterstützt, dies in der Praxis allerdings nicht angewandt werden kann, weshalb fragwürdig ist, ob nicht mit den bewährten Methoden des Übersetzerbaus – Definitionstabellen und Attributgrammatiken (siehe [16]) – dasselbe Ziel erreicht werden kann.

Die Arbeiten [24, 11] und [12] beschreiben neuere Ansätze, die mit Methoden aus dem Modell-basierten Umfeld gelöst werden sollen und welche Probleme auftreten. So beschreibt [24] ein Modell auf Basis der Eclipse Modeling Projects, bei dem sich die Autoren eine Methode wünschen um Semantik und Optimierungen in das Meta-Modell zu integrieren. In [11] wird über ein aktuelles Forschungsprojekt berichtet, bei dem bisher vor allem die Elemente eines Meta-Modells, welches in XML umgesetzt werden soll, aufgeführt werden. Die Autoren beschreiben nicht ob und wie eine Semantik des eigentlichen Modells und des dazugehörigen Meta-Modells aussehen soll. Dass XML-Formate ohne Analysen und ohne Einbettung domänenspezifischen Wissens in das Meta-Modell zu vielen inkonsistenten Daten führen, zeigt [1] recht anschaulich: das „zentrale Waffenregister für Deutschland“ ist aus Sicht der Polizeigewerkschaft nicht nutzbar, da Daten scheinbar ohne Prüfung eingegeben werden konnten. In [12] wird beschrieben, welche Evolutionsschritte (Änderungen) ein Modell zum Variantenmanagement enthalten muss, d.h. die Anwendungsfälle werden beschrieben, und dass komplexere Evolutionsschritte in Zukunft automatisiert implementiert werden sollen. In [8] werden eine Reihe von Herausforderungen aufgeführt, die notwendig sind um von einem Problem zur Implementierung mit Hilfe von Modellen zu kommen. Da in [24] noch immer nach der formalen Beschreibung der Semantik von Modellen gesucht wird, ist davon auszugehen, dass [8] weiterhin aktuell ist. Einige der Herausforderungen können also mit den Methoden des Übersetzerbaus gelöst werden. Wir gehen davon aus, dass auch obige Arbeiten ([11, 12, 24]) von einer Nutzung der Methoden aus dem Übersetzerbau profitieren können.

Eine Arbeit, die etwas aus dem Rahmen fällt ist [21], denn hier wird ein Meta-Modell zur Analyse von geographischen Daten beschrieben, wobei die Analysekriterien mittels OCL ausgedrückt werden. Das Besondere an dieser Arbeit ist, dass dieses Meta-Modell und die dazugehörige Analyse dann in eine attributierte Grammatik umgesetzt wurde. Die Autoren haben somit erste Schritte aufgezeigt die eine automatisierte Umwandlung von Meta-Modellen und OCL in attributierte Grammatiken ermöglichen können.

7 Zusammenfassung

In dieser Arbeit haben wir das Grundproblem unseres Forschungsprojekts und einige der damit verbundenen Probleme vorgestellt. Gleichzeitig haben wir gezeigt, dass die Methoden des Übersetzerbaus geeignet sind einen ersten Ausschnitt aus dem Prozess abzubilden. Unserem Prüfverfahren liegt das bekannte Ver-

[7] engl. Reference Attribute Grammar (RAG)

fahren Tiefensuche zugrunde, beschrieben unter anderem in [4]. Mögliche Anwendungen der Tiefensuche finden sich in [13]. Zusammen mit den Methoden des Übersetzerbaus [9, 16] unter Verwendung attributierter Grammatiken[15] war es uns möglich die Konsistenzprüfungen wesentlich schneller durchzuführen als dies mit äquivalenten Methoden, die im Modell-basierten Umfeld vorgeschlagen werden, möglich ist.

Wir haben gezeigt, dass auch ohne Nutzung von NLP Stellen der Inkonsistenz in Anforderungsschlagworten, einer Form von Anforderungen, auffindbar sind. Wir haben begründet und ggf. auf andere Stellen verwiesen, warum die Methoden des Übersetzerbaus für unsere Zwecke notwendig sind und die Methoden, die bisher im Modell-basierten Umfeld verwendet werden nicht ausreichen.

Das Requirements Interchange Format (ReqIF), auf das sich [6] bezieht begründet auch warum ReqIF nicht ausreichend ist, aber als Austauschformat nützlich ist. Wir planen ReqIF als Austauschformat zu unterstützen.

Weiterhin gilt es zu erforschen wodurch die Trägheit von XText Zustande kommt und welche Möglichkeiten der Verbesserung sich hier anbieten. Auch muss geprüft werden ob dies für andere textuelle Sprachen unter Verwendung anderer Werkzeuge, wie EMFText, ebenso der Fall ist.

In weiteren Schritten wollen wir die Spezifikation von Anforderungen und Konfigurations- sowie Variantenmanagement hinzufügen. Hierbei können wir auf den bekannten Methoden aus [23] und [3] aufbauen. Weiter offen sind dann jene Punkte, die unter anderem [20] und [8] als Herausforderungen markieren: Konsistenzprüfung, Traceability und mehrere Sichten auf dasselbe Modell.

Danksagung
Wir danken unseren Projektpartnern für die gute Zusammenarbeit und die Bereitstellung umfangreicher Beispiele aus dem aktuellen Arbeitsalltag. Weiterhin danken wir dem Projektträger VDI/VDE-IT und dem BMBF, die diese Arbeit im Rahmen des Projekts ELSY (Nr. 16M3202D) fördern.

Literatur

[1] Borchers, Detlef: *Polizeigewerkschaft: Waffenregister ist „Schuss in den Ofen“*. http://www.heise.de/newsticker/meldung/Polizeigewerkschaft-Waffenregister-ist-Schuss-in-den-Ofen-1937209.html, August 2013

[2] Bürger, Christoff ; Karol, Sven ; Wende, Christian ; Assmann, Uwe: Reference Attribute Grammars for Metamodel Semantics. Version: 2011. http://dx.doi.org/10.1007/978-3-642-19440-5_3. In: Malloy, Brian (Hrsg.) ; Staab, Steffen (Hrsg.) ; Brand, Mark (Hrsg.): *Software Language Engineering* Bd. 6563. Springer, 2011. – DOI 10.1007/978–3–642–19440–5_3. – ISBN 978–3–642–19439–9, S. 22–41

[3] Classen, Andreas ; Heymans, Patrick ; Schobbens, Pierre-Yves: What's in a feature: A requirements engineering perspective. Version: 2008. http://dx.doi.org/10.1007/978-3-540-78743-3_2. In: *Fundamental Approaches to Software Engineering*. Springer, 2008. – DOI 10.1007/978–3–540–78743–3_2, S. 16–30

[4] Cormen, Thomas H. ; Leiserson, Charles E. ; Rivest, Ronald. ; Stein, Clifford: *Algorithmen-Eine Einführung*. Oldenbourg Verlag, 2004

[5] Costal, Dolors ; Gómez, Cristina ; Queralt, Anna ; Raventós, Ruth ; Teniente, Ernest: Facilitating the definition of general constraints in UML. Version: 2006. http://dx.doi.org/10.1007/11880240_19. In: *Model Driven Engineering Languages and Systems*. Springer, 2006. – DOI 10.1007/11880240_19, S. 260–274

[6] Ebert, Christof ; Jastram, Michael: ReqIF: Seamless requirements interchange format between business partners. In: *Software, IEEE* 29 (2012), Nr. 5, S. 82–87. http://dx.doi.org/10.1109/MS.2012.121. – DOI 10.1109/MS.2012.121

[7] Ebert, Jürgen ; Franzke, Angelika: A declarative approach to graph based modeling. In: *Graph-Theoretic Concepts in Computer Science* Springer, 1995, S. 38–50

[8] France, Robert ; Rumpe, Bernhard: Model-driven Development of Complex Software: A Research Roadmap. In: *2007 Future of Software Engineering*. Washington, DC, USA : IEEE Computer Society, 2007 (FOSE '07). – ISBN 978–0–7695–2829–8, S. 37–54

[9] Gray, Robert W. ; Levi, Steven P. ; Heuring, Vincent P. ; Sloane, Anthony M. ; Waite, William M.: Eli: A complete, flexible compiler construction system. In: *Communications of the ACM* 35 (1992), Nr. 2, S. 121–130. http://dx.doi.org/10.1145/129630.129637. – DOI 10.1145/129630.129637

[10] Heidenreich, Florian ; Johannes, Jendrik ; Karol, Sven ; Seifert, Mirko ; Thiele, Michael ; Wende, Christian ; Wilke, Claas: Integrating OCL and Textual Modelling Languages. Version: 2011. http://dx.doi.org/10.1007/978-3-642-21210-9_34. In: Dingel, Juergen (Hrsg.) ; Solberg, Arnor (Hrsg.): *Models in Software Engineering* Bd. 6627. Springer, 2011. – DOI 10.1007/978–3–642–21210–9_34. – ISBN 978–3–642–21209–3, S. 349–363

[11] Hess, Steffen ; Gross, Anne ; Maier, Andreas ; Orfgen, Marius ; Meixner, Gerrit: Standardizing model-based in-vehicle infotainment development in the German automotive industry. In: *Proceedings of the 4th International Conference on Automotive User Interfaces and Interactive Vehicular Applications*. New York, NY, USA : ACM, 2012 (AutomotiveUI '12). – ISBN 978–1–4503–1751–1, S. 59–66

[12] Holdschick, Hannes: Challenges in the evolution of model-based software product lines in the automotive domain. In: *Proceedings of the 4th International Workshop on Feature-Oriented Software Development*. New York, NY, USA : ACM, 2012 (FOSD '12). – ISBN 978–1–4503–1309–4, S. 70–73

[13] Hopcroft, John E. ; Tarjan, Robert E.: Efficient algorithms for graph manipulation. Version: 1971. http://dx.doi.org/10.1145/362248.362272. Stanford, CA, USA : Stanford University, 1971 (CS-TR-71-207). – Forschungsbericht. – ftp://reports.stanford.edu/pub/cstr/reports/cs/tr/71/207/CS-TR-71-207.pdf

[14] Johnson, Ralph ; Helm, Richard ; Vlissides, John ; Gamma, Erich: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995. – ISBN 978–0–201–63361–0

[15] Kastens, Uwe: Ordered attributed grammars. In: *Acta Informatica* 13 (1980), Nr. 3, S. 229–256. http://dx.doi.org/10.1007/BF00288644. – DOI 10.1007/BF00288644

[16] Kastens, Uwe. ; Waite, William M.: An abstract data type for name analysis. In: *Acta Informatica* 28 (1991), Nr. 6, S. 539–558. http://dx.doi.org/10.1007/BF01463944. – DOI 10.1007/BF01463944. – ISSN 0001–5903

[17] Oesterle, Manfred (Hrsg.) ; Leidig, Fred (Hrsg.): *Methodisch sichere, schnelle Produktionsanläufe in der Mechatronik (MESSPRO) - Band 2 der Reihe „Schneller Produktionsanlauf in der Wertschöpfungskette“*. Frankfurt : VDMA-Verlag, 2007. – ISBN 978–3–8163–0550–7

[18] Pandey, R. K.: Object constraint language (OCL): past, present and future. In: *SIGSOFT Softw. Eng. Notes* 36 (2011), Januar, Nr. 1, S. 1–4. http://dx.doi.org/10.1145/1921532.1921543. – DOI 10.1145/1921532.1921543. – ISSN 0163–5948

[19] Partsch, Helmut A.: *Requirements Engineering systematisch*. Springer, 2010. http://dx.doi.org/10.1007/978-3-642-05358-0. http://dx.doi.org/10.1007/978-3-642-05358-0

[20] Pretschner, Alexander ; Broy, Manfred ; Kruger, Ingolf H. ; Stauner, Thomas: Software engineering for automotive systems: A roadmap. In: *2007 Future of Software Engineering* IEEE Computer Society, 2007, S. 55–71

[21] Schmittwilken, Jörg ; Saatkamp, Jens ; Forstner, W. ; Kolbe, Thomas H. ; Plumer, L.: A semantic model of stairs in building collars. In: *PHOTOGRAMMETRIE FERNERKUNDUNG GEOINFORMATION* 2007 (2007), Nr. 6, S. 415

[22] Schwarz, Hannes ; Ebert, Jürgen ; Winter, Andreas: Graph-based traceability: a comprehensive approach. In: *Software & Systems Modeling* 9 (2010), Nr. 4, S. 473–492. http://dx.doi.org/10.1007/s10270-009-0141-4. – DOI 10.1007/s10270–009–0141–4

[23] Shaker, Pourya: Feature-oriented requirements modelling. In: *2010 ACM/IEEE 32nd International Conference on Software Engineering* Bd. 2 IEEE, 2010, S. 365–368

[24] Sun, Yu ; Gray, Jeff ; Bulheller, Karlheinz ; Baillou, Nicolaus von: A model-driven approach to support engineering changes in industrial robotics software. In: *Proceedings of the 15th international conference on Model Driven Engineering Languages and Systems*. Berlin, Heidelberg : Springer, 2012 (MODELS'12). – ISBN 978–3–642–33665–2, S. 368–382

[25] Waite, William ; Kastens, Uwe ; Sloane, Anthony M.: *Generating Software from Specifications*. USA : Jones and Bartlett Publishers, Inc., 2007. – ISBN 978–0–7637–4124–2

COMPOSITA: A Study in Runtime Architectures for Massively Parallel Systems

Luc Bläser[1] and Jürg Gutknecht[2]

[1] University of Applied Sciences Rapperswil
Institute for Software
lblaeser@hsr.ch
[2] ETH Zürich, Switzerland
Native Systems Group
gutknecht@inf.ethz.ch

Abstract. COMPOSITA is an experimental operating system optimized for effective multi-processing and memory management. Based on new software technology, notably including micro-stacks and software-controlled preemption, the system supports millions of concurrent light-weight processes. Thanks to hierarchical component structures, no garbage collection is used for the management of dynamic memory. Experimental evaluations have shown that, under the governance of COMPOSITA, massively concurrent programs perform and scale considerably better than under traditional operating systems.

1 Introduction

A comparison of today's most popular operating systems, notably *Windows*, *Linux*, *MacOSX* etc. shows a striking principal resemblance of their architectures. This is beyond sheer coincidence because all of these systems share the same genetic footprint: the UNIX systems of the 1960s and 1970s. While the success of these UNIX derivatives is undeniably remarkable, their efficiency factor on modern computer systems cannot possibly be close to optimal, as their genes in essence favor a totally different scenario of usage: a central system serving a large number of terminal users in time-sharing mode (or a miniaturized version hereof).

Ever more challenging requirements like parallelism across all granularities, highly concurrent data structures and seamlessly integrated real-time applications exert additional pressure on system architects towards rethinking the entire software stack of a modern computer system from the ground up. We decided to take up this challenge in the concrete form of building an super-efficient implementation of a runtime environment for a highly parallel programming language on a state-of-the-art multicore machine. CL became the language of our choice, an experimental component-oriented language that we developed in a previous project [4]. CL embodies a model of fine-granular concurrency, message communication and hierarchical composition. The result of our endeavor is an ultra-compact operating system called COMPOSITA. It is targeted at massively parallel applications and exhibits a variety of innovative features, among them

- **Fine-grained call-stacks** (micro-stacks) permitting millions of concurrent light-weight processes to coexist at any time
- **Super-fast context switches**, equally efficient as procedure calls
- **Software-instrumented checkpoints** as a replacement for expensive preemptive context switches
- **Ultra-compact process backups** exploiting context intelligence of the compiler
- **Memory management and recycling** based on hierarchical compositions instead of garbage collection

Our decision of developing a new and self-contained solution from the ground up (in contrast to fixing and extending an existing system) allowed us to concentrate on the essentials (thereby sacrificing some "bells and whistles") and to keep it compact and simple. The research was primarily meant to provide an experimental ground for exploring new models and much less to give a definitive answer on how generic operating systems should be built in the future.

In the following, we shall explain the design and the detailed implementation of the COMPOSITA operating system. Section 2 gives an overview of its architecture, including the underlying component-oriented programming language CL. Section 3 focuses on the management of processes and memory. Section 4 reports on some experimental measurements. Section 5 discusses related work and Section 6 finally draws conclusions.

2 Architecture

COMPOSITA runs on Intel PC machines, has a size of ca. 250 KB and boots up in about a second. It currently includes a limited set of device drivers, notably comprising IDE disk, keyboard and graphics card. Both the system and the applications present themselves in the form of a network of *components* interconnected via *interfaces* (Figure 1).

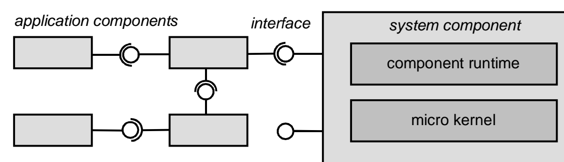


Fig. 1. System architecture

2.1 Component Structures

The component architecture is based on the following mechanisms and principles:

- **Composites.** Each component may hierarchically contain any number of sub-components.
- **Interfaces.** Interfaces specify communication ports. Each component potentially features two kinds of interfaces: offered interfaces and required interfaces.
- **Connections.** If compatible, a required interface of one component can be connected with an offered interface of another component, where compatibility means that the interfaces have the same name and reside in the same surrounding composite.
- **Delegation.** A component can delegate an offered interface of its own to an offered interface of one of its sub-components. And analogously for required interfaces. In both cases the interface names are required to match.

Figure 2 depicts an exemplary composite called `FileConverter`, where the offered/ required interfaces are represented as lollipops/ forks respectively. The wiring comprises connecting the required interface `Reader` of the `Transformer` with the offered interface `Reader` of the `Parser` and delegating the offered interface `ConversionControl` to the offered interface of the `Transformer`.

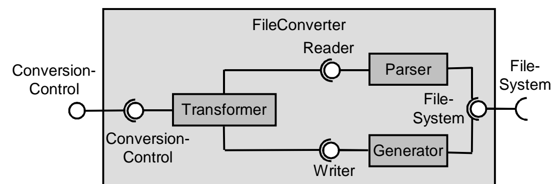


Fig. 2. Exemplary component structure

Components are entirely managed by their container (surrounding component), i.e. only an outer component can create, connect and delete inner components. All other kinds of interactions necessarily involve communication (via interfaces).

2.2 Concurrency Model

Concurrency in CL interoperates with the compositional structure in the following way:

- **Processes and internal synchronization.** A component typically runs a number of concurrent processes inside of its scope, where each process is allowed to access the component's (shared) memory. For the purpose of synchronization both exclusive locks and shared locks are provided, as well as a generic wait (Boolean condition) operation [11].

- **Communication.** Interfaces allow bidirectional communication. The message format as well as the protocol supported by a certain interface is specified in terms of a formal grammar bound to the respective interface. Messages may include numbers, strings, etc. but also components (transported from sender to receiver). Communications between each pair of connected components run separately and in parallel.

3 Operating System

The COMPOSITA system supports dynamic loading of components. It presents itself as a pre-existing component loaded and started after system boot up. It makes available system services and device drivers to application components via interfaces and the communication mechanism discussed earlier. In the following sections we explain the implementation of the framework behind the scenes of these concepts in relevant detail.

3.1 Process Management

As our system is aimed at accommodating a very large number of concurrent processes, the (footprint) size of individual processes is an important factor. Therefore, we employ special care to minimize a) the stack size and b) the backup space needed (after context switch) for each individual process.

Process Stacks Unlike in many other operating systems, process stacks are not implemented as contiguous pre-allocated memory blocks of a certain (often generously guessed) maximum size in some virtual address domain in COMPOSITA but as a linear list of heap blocks in real memory, with the immediate benefit that stacks can dynamically grow or shrink, see Figure 3. At each procedure call a runtime check is instrumented to determine whether sufficient stack space is still present. If not, a new stack block of a precalculated size (by the compiler) is allocated, and the stack grows correspondingly. When the procedure later returns, the extra stack block is deallocated and returned to the heap. The size of a process' initial stack block is also determined by the compiler.

It is worth noting that CL does not require frequent dynamic stack extensions as it uses communication-based component interactions in place of method calls in all but local cases. As an additional precaution interrupts run in privileged mode, using a separate, preallocated kernel stack.

Context Switches The efficiency of *context switches* is a decisive measure for the performance of any highly parallel system. A *synchronous* context switch occurs whenever a command of type *wait/suspend* is to be executed. As our kernel does not run in a separate protection ring, no (expensive) software interrupt is needed to handle this case and an ordinary procedural kernel call (prolog and epilog operating on different stacks though) does the job. Only three registers

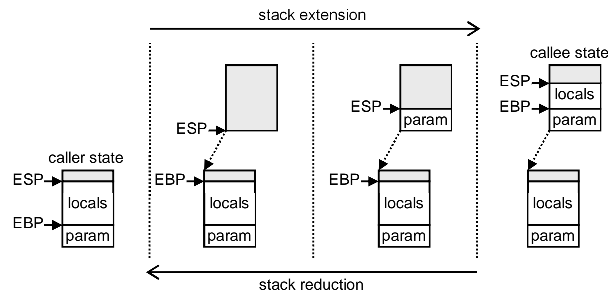


Fig. 3. Stack extension and reduction

must be saved and restored namely the *program counter*, the *stack pointer* and the *frame pointer*.

Asynchronous context switches are indispensable with runtime strategies employing *preemptive process scheduling*, such as for example preemptive priority scheduling or processor-sharing based on the allocation of time-slices. As asynchronous context switches are uncooperative, they tend to be highly expensive both in terms of time and space (request for saving the entirety of registers including FPU, MMX, SSE2). Therefore, we were looking for an alternative and more efficient solution that allows COMPOSITA to mimic preemption in a slightly weaker form. Rather than letting hardware interrupts govern the context switch, we are using an instrumented inline approach. More concretely, the CL compiler injects checks for pending preemption requests directly into the machine code at suitable locations, where suitable means that the runtime can guarantee at least one check within each time-interval of a desired length. Furthermore, the compiler can quite easily optimize the locations in a way to minimize the number of registers to be saved when the context switch actually occurs, which substantially contributes to compact sizes of process backups.

According to our heuristics, the compiler injects a check for preemption requests (1) in each loop body, (2) in each procedure entry, and (3) after a statement sequence of a maximum (worst-case) runtime. The current implementation of a check only requires a few simple instructions. Table 1 quantifies the performance costs of preemption checks for sample programs. For typical concurrent scenarios, no or very low overheads can be observed. Of course, costs become more significant if programs involve hot loops or frequent procedure calls: The artificial spinning loop example shows approximately 20% overheads for components running long counting loops.

Process Scheduling All runnable (including preempted) processes are queued up in a single FIFO list (ready list) and assigned to processors by a simple round-robin scheduler. Components are implemented as monitor-protected shared resources. Each component features two waiting lists, one for processes waiting for an exclusive lock or shared lock and one for processes waiting on a Boolean

Program	With checks	No checks	Overheads
ProducerConsumer (1 producer, 1 consumer, capacity 1)	1011	1048	-3.5%
Eratosthenes (10000 upper limit)	234	230	1.9%
TokenRing (1000 nodes)	266	273	-2.7%
SpinningProcesses (16 instances)	181	151	19.6%

Runtime in milliseconds, rounded to millisecond, average of 3 subsequent executions,
Intel 2 Core i7 3520M, 2.9GHz, 8GB main memory.

Table 1. Runtime costs of preemption checkpoints

condition. The prioritization among processes follows a so-called eggshell model [11] that grants priority handling to waiting processes whose condition has be established over newly entering processes. This is implemented by checking the waiting lists for established conditions whenever a process releases the monitor lock (signal and exit strategy). If an established condition has been found, the lock is immediately passed to the corresponding process. In order to avoid potential starvation among reader or writer processes, a first-come first-served strategy is applied to the processes entering a lock-controlled region.

Communication Channels The formal specification of communication protocols in CL empowers the runtime system to validate the implementation of protocols at both ends, thereby upgrading the level of intercomponent communication in CL in terms of consistency checking to the level of method calls in strongly typed languages. A state machine derived from the protocol specification is used for this purpose. In principle, static consistency checking of protocol implementations would also be possible up to a certain degree, albeit a complete static analysis is impossible because the problem is equally hard as the halting problem that is undecidable. Implementation-wise, communication channels in CL are based on small bounded buffers (maximum of 4 messages).

3.2 Memory Management

Perhaps the biggest pay-out of our strictly hierarchical component system is a drastically simplified memory management. Traditional systems maintain a non-hierarchical object graph. Lacking any natural ownership relation, a sophisticated system-wide process is put into action with the mandate of continuously identifying garbage (objects to be recycled) via *reachability*. In COMPOSITA, *containment* defines a natural ownership relation that authorizes containers (the owners) to finalize, deallocate and recycle their content components explicitly. In detail, deallocation of component C involves the following steps (see Figure 4 for an illustration):

1. Recursively delete (inner) components of C

2. Wait for all communications involving C to terminate
3. Disconnecting interfaces with C

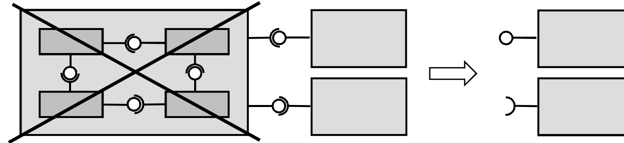


Fig. 4. Component deletion

It is worth noting that our approach does not suffer from the traditional problems of delayed finalization or from phenomena like resurrection etc., nor is the consistency of the memory compromised at any time. Dangling pointers do not exist, as external accesses to components never involve pointers but communication instead and as any message traffic is suspended after the interface has been disconnected. Memory leaks are impossible as well because components remain accessible (by their container) until they are deallocated explicitly. No component ever becomes undeletable.

4 Experimental Results

The COMPOSITA system was uncompromisingly designed with the single goal in mind of running an extremely large number of concurrent processes that is clearly beyond the capabilities of existing systems. Correspondingly, our first benchmark focuses on comparative measurements of the maximum number of (light-weight) processes that can be accommodated concurrently (on top of the given hardware) by COMPOSITA and other systems respectively. For this purpose, the benchmark creates and runs fake processes that merely execute a simple waiting activity. Table 2 summarizes the results. As can be seen, COMPOSITA is able to accommodate millions of processes, while the classical systems reach their limits at a lower order of magnitude. This can quite easily be explained by COMPOSITA's scaling characteristics that are by design inversely linear with regard to the stack size and linear with regard to the size of the physical memory.

For measuring the runtime performance we assembled a small set of programs representing typical concurrency patterns, all of them rich in terms of context switches: Producers/ Consumers (N producers and M consumers interacting via a bounded buffer of capacity C), Sieve of Eratosthenes (a pipeline of filtering processes determining the prime numbers between 2 and an upper limit N) and TokenRing (a ring of N processes circularly ordered and pushing a token around the ring 1000 times). In COMPOSITA, component interactions are modeled as message communications with an extra service process per channel. In all other languages, conventional method calls are used for this purpose. All measurements

were performed on an Intel 2 Core i7 3520M 2.9GHz platform, with 8GB main memory. Table 3 clearly demonstrates that our architecture outperforms classical thread-based systems, mostly due to COMPOSITA's highly optimized context switches.

COMPOSITA	.NET (Win8)	Java (Win8)
4,367,000	100,000	100,000

Number of light-weight processes under COMPOSITA, number of threads under .NET x64 and Java 64 bit VM, 8GB main memory, rounded on 3 figures, .NET Framework 4.5 under Windows 8 (x64), Java 7 (1.7.0.21, 64 bit server VM) under Windows 8 (x64).

Table 2. Maximum number of threads

Program	COMPOSITA	C#	Java
ProducerConsumer (N=M=C=1)	**1011**	5427	6020
ProducerConsumer (N=1, M=10, C=1)	**1327**	22324	26255
ProducerConsumer (N=10, M=10, C=10)	**10141**	40158	30513
Eratosthenes (N=1000)	**5**	31	31
Eratosthenes (N=10'000)	**235**	877	640
Eratosthenes (N=100'000)	**14594**	49216	38023
TokenRing (N=1000)	**266**	4500	4596
TokenRing (N=10'000)	**2822**	49945	53582
TokenRing (N=100'000)	**30106**	518956	1,163,088

Runtime in milliseconds, rounded to millisecond, average of 3 subsequent executions, Intel 2 Core i7 3520M, 2.9GHz, 8GB main memory. C# on .NET Framework 4.5, x64, with optimization compiler option, on Windows 8, Java 7 version 1.7.0.21, 64 bit server under Windows 8.

Table 3. Runtime of concurrent programs

5 Related Work

Stack architectures. Several systems share with COMPOSITA the representation of call-stacks as dynamic lists of memory blocks with the goal of to reducing the memory footprint of thread representations [1, 9]. However, unlike COMPOSITA's micro-stacks, the stack size in these systems still remains roughly at

page granularity [9, 14]. Even lighter-weight threads are usually less useful to these systems because context switches such as the ones needed for preemptive multitasking [14, 12] are no longer possible. Such restrictions apply by purpose for coroutines and fibers. Other implementations apply stack hijacking that is they require the system to back up the stack contents at each context switch, so to allow the new thread to continue on the same stack [3]. Lazy thread creation [8] optimizes certain concurrent executions by sequentializing them. Light-weight thread APIs and thread pools are standard facilities provided by most parallel system implementations. However, the formers usually suffer from the need of mapping user threads to rather inefficient kernel threads (for truly parallel execution) [14, 3], while the latters come at a cost of a de facto restricted applicability to threads with no mutual dependencies.

Garbage collection. Garbage collection in real-time systems has become a pain in the neck. Extremely complicated and subtle algorithms have been conceived with the goal of eliminating or at least reducing the non-deterministic disruptions caused by garbage collection [7, 1, 13], without having yet achieved a truly satisfactory solution. Other approaches employ multiple isolated object spaces, managed by separate garbage collectors (see for example Singularity OS). COMPOSITAs memory management rests on the belief that eliminating the problem is its best solution.

Composita. An earlier version of the system has been outlined in [5, 6].

6 Conclusion

We have invested a substantial research effort into the question of how a modern runtime system architecture, optimally supporting massive parallelism and free of legacy should or could look like. Our strategy was radical: develop a prototype from the ground up as a proof of concept. The result is COMPOSITA, a kind of a mockup of a fully grown system and uncompromisingly targeted at running an extremely large number of concurrent processes. We consider our experiment a success. Thanks to a number of innovative solutions, including hierarchical memory structures, micro-stacks, low-cost context switches and code-instrumented preemption, we are able to convincingly demonstrate that both the performance and the scalability of massive parallel hardware systems can be substantially improved if orchestrated by cleverly designed software.

Project Website

The COMPOSITA system and all test programs used for measurements are available at: http://concurrency.ch/Projects/Composita

Acknowlegments

We gratefully acknowledge the collaboration with Kai Nagel. Thanks to his excellent support and advice we were not only able to realize a realistic traffic

simulation on top of COMPOSITA but in addition to practically demonstrate its performance advantage over traditional simulation systems [6]. During this project, Felix Friedrich and Svend Knudsen continuously and patiently provided most helpful input and valuable feedback on both the design and the implementation of our system.

References

1. D. F. Bacon, P. Cheng, and V. T. Rajan. *A Real-Time Garbage Collector with Low Overhead and Consistent Utilization.* Proceedings of the Symposium on Principles of Programming Languages (POPL), January 2003.
2. R. von Behren, J. Condit, F. Zhou et al. *Capriccio: Scalable Threads for Internet Services.* Proceedings of the 19th ACM symposium on Operating systems principles (SOSP), October 2003.
3. A. Begel, J. MacDonald, and M. Shilman. *PicoThreads: Lightweight Threads in Java.* Technical Report, UC Berkeley, 2000.
4. L. Bläser. *A Component Language for Structured Parallel Programming.* Proceedings of the Joint Modular Languages Conference (JMLC), Oxford, UK, September 2006.
5. L. Bläser. *A High-Performance Operating System for Structured Concurrent Programs.* Proceedings of the Workshop on Programming Languages and Operating Systems (PLOS), October 2007.
6. L. Bläser. *A Component Language for Pointer-Free Concurrent Programming and its Application to Simulation.* PhD Thesis, Diss. ETH No. 17480, ETH Zurich, November 2007.
7. P. Cheng and G. E. Blelloch. *A Parallel, Real-Time Garbage Collector.* Proceedings of the Conference on Programming Language Design and Implementation (PLDI), June 2001.
8. S. C. Goldstein, K. E. Schauser, and D. E. Culler. *Lazy Threads: Implementing a Fast Parallel Call.* Journal of Parallel and Distributed Computing, 37: 5-20, 1996.
9. G. C. Hunt and J. R. Larus. *Singularity: Rethinking the Software Stack.* ACM SIGOPS Operating Systems Review, 41(2): 37-49, April 2007.
10. G. Hunt, J. Larus, M. Abadi, et al. *An Overview of the Singularity Project.* Technical Report MSR-TR-2005-135, Microsoft Research, October 2005.
11. P. J. Muller. *The Active Object System: Design and Multi-processor Implementation.* PhD Thesis, Diss. ETH No. 14755, ETH Zurich, 2002.
12. E. D. Polychronopoulos, X. Martorelli, D. S. Nikolopoulos et al. *Kernel-Level Scheduling for the Nano-Threads Programming Model.* Proceedings of the 12th International Conference on Supercomputing (ICS), Melbourne, Australia, July 1998.
13. Y. Sun and W. Zhang. *Overview of Real-Time Java Computing.* Journal of Computing Science and Engineering 7.2 2013: 89-98, 2013.
14. K. B. Wheeler, R. C. Murphy and D. Thain. *Qthreads: An API for Programming with Millions of Lightweight Threads.* Proceedings of the 22nd IEEE International Symposium on Parallel and Distributed Processing (IPDPS), April 2008.

Ein auf partieller Auswertung basierendes Tool-Framework für binäre Datentypen

Stefan Bohne
senTec Elektronik GmbH, Ilmenau
stefan.bohne@sentec-elektronik.de

Prof. Dr. rer. nat. Baltasar Trancón Widemann (JP)
Fakultät für Informatik und Automatisierung, TU Ilmenau
baltasar.trancon@tu-ilmenau.de

25. August 2013

Abstract

Parsing and generation of bit strings is on some layer a fundamental part of all computing tasks. Such objects can be seen as binary data types. Several approaches of formalizing such data types and using them to aid programmers have been made.

We analyze some of them and show problems in their design. We then propose to use partial evaluation as the means to provide a very flexible and efficient way to generate code for handling binary data structures.

1 Einleitung

Auf der niedrigsten Abstraktionsschicht lassen sich alle Programme als Manipulation von Bitfolgen interpretieren. So ist ein Compiler für eine Programmiersprache im Prinzip nichts anderes als eine Funktion, die eine Bitfolge, den Programmquelltext, in eine andere Bitfolge, die ausführbare Datei, umwandelt. In höheren Programmiersprachen wird diese Sichtweise fast vollständig durch komplexere Abstraktionen ersetzt, um dem Programmierer die Last der Umformung zwischen Bitfolge und Interpretation zu ersparen. So ist z.B. die binäre Folge 01000000 01001001 00001111 11011011 (oder 40 49 0F DB in Hexadezimal) nur schwer als Fließkommazahl mit dem Wert 3,141593 zu erkennen. Wir sind diese Darstellung von Fließkommazahlen auch nicht gewohnt — der Compiler erledigt diese Transformation immer für uns.

Trotzdem tauchen Bitfolgen als Programmierelemente auch in höheren Programmiersprachen noch auf, typischerweise, wenn es darum geht beliebige Daten zu transportieren, wie z.B. beim Lesen und Schreiben auf die Festplatte oder

```
// Lese e aus b
e = (b >> 23) & 0xFF;
// Schreibe e in b
b = (b & ~(0xFF << 23)) | ((e & 0xFF) << 23);
```

Abbildung 1: Lesen und Schreiben des Exponenten einer Fließkommazahl, ausgedrückt in Ganzzahloperationen

beim Senden und Empfangen von Netzwerkpaketen. An solchen Stellen muss der Programmierer die Relation zwischen komplexeren Datenstrukturen und deren Entsprechung als Bitfolge herstellen, d.h. die Bits müssen geparst bzw. generiert werden. Die Konvertierung zwischen Objekten und Bitfolgen wird auch häufig Serialisierung genannt.

In der embedded und hardwarenahen Programmierung hat man meist viel mehr mit direkten Manipulation von Bits zu tun, da hier die Abstraktionen von (noch) höheren Programmiersprachen fehlen. Außerdem ist beim direkten Zugriff auf Hardwareregister aufgrund ihrer Struktur ein sogenanntes Bit-twiddling unumgänglich.

Fließkommazahlen mit einfacher Präzision sollen hier als Beispiel dienen. Sie lassen sich als Tupel $(s : \{+1, -1\}, m : [0..2^{23} - 1], e : [-126..127])$ aus Vorzeichen (s), Mantisse (m) und Exponent (e) betrachten. Der Zahlenwert berechnet sich dann aus $s \cdot (1 + \frac{m}{2^{23}}) \cdot 2^e$. Sonderfälle, mit deren Hilfe 'null', 'unendlich' und 'undefiniert' dargestellt werden, werden hier zugunsten der Lesbarkeit nicht betrachtet. Im Folgenden soll anhand dieser Tupeldarstellung aufgezeigt werden, welche Schwierigkeiten der direkte Umgang mit Bitfolgen in der heutigen Softwareentwicklung mit sich bringt.

2 Binäre Datentypen in Programmiersprachen

In den meisten Programmiersprachen kann man das Layout der Datenstrukturen im Arbeitsspeicher nicht oder nur wenig beeinflussen. Das ist für die meisten Anwendungen auch der richtige Weg, da so dem Compiler mehr Raum für Optimierungen gelassen wird. Also bleibt nichts weiter übrig, als Datenstrukturen in einem separaten Schritt zwischen der Höheren und der Bit-Darstellung zu konvertieren.

Hier bieten die populären Programmiersprache wie C/C++, Java, C#, Python keine Hilfsmittel. Die am besten geeigneten Datentypen um Bit-Darstellungen zu repräsentieren sind die maschinennahen Ganzzahltypen (`int`) für kleinere Daten und Folgen von Bytes (`char*`, `byte[]`, `bytes`) für größere Daten oder Daten variabler Länge. Das heißt, dass die Algorithmen für die Konvertierung zwischen den Darstellungen auch nur in den sehr maschinennahen Operationen ausgedrückt werden können, die von diese maschinennahen Ganzzahl-Datentypen bereitgestellt werden. Abbildung 1 zeigt in Pseudocode, wie das Lesen und Schreiben des Exponenten einer Fließkommazahl typischerweise aussieht. Insbe-

```
# Lese e aus b
e = b[23:31].uint
# Schreibe e in b
b[23:31] = Bits(uint=e, length=8)
```

Abbildung 2: Lesen und Schreiben des Exponenten einer Fließkommazahl, ausgedrückt mit der python-bitstring Bibliothek [4]

sondere das Schreiben ist recht komplex. Da es nicht möglich ist nur Teile einer Zahl zu setzen, werden zunächst die entsprechenden Bits mit einem bitweisen Und (`&`) auf Null gesetzt und danach der neue Teil mit einem bitweisen Oder (`|`) verknüpft. In maschinennahem Code oder in Code, der sich mit optimierten Datenformaten beschäftigt, kann man solche Codefragmente in großer Zahl finden.

Eigentlich sind für unser Beispiel nur zwei Parameter relevant: die Länge der Komponente in Bits (hier 8) und die Position der Komponente (hier ab Bit Nr. 23). Die Quelltext, die so entstehen, haben trotzdem ziemlich schlechte Eigenschaften.

1. Die Parameter tauchen in den Teilausdrücken mehrfach auf und häufig nicht in Reinform. Im Beispiel taucht die Bitlänge nur als Bitmaske `0xFF` auf, was eine Vereinfachung des Ausdrucks `(1 << 8) - 1` ist.

2. Die Fragmente sind meist nur eine Zeile lang, sodass sich der Aufwand diese in eigene Funktion zu refaktorieren meist (scheinbar) nicht lohnt.

3. Lesen und Schreiben sind dadurch meist im Quelltext voneinander entfernt, sodass man nicht leicht erkennen kann, ob sie konsistent miteinander sind.

4. Der Quelltext ist alles andere als selbstdokumentierend, was dazu führt, dass ein weiteres Dokument zur Dokumentation des Bitformats angelegt werden muss, welches dann wieder inkonsistent zum Quelltext sein kann (und es in der Praxis meistens auch ist).

Den ersten Schritt, die Situation zu verbessern, ist, einen höheren Datentyp für Bitfolgen bereitzustellen, mit dem sich die typischen Operationen einfacher schreiben lassen. Für die meisten höheren Programmiersprachen gibt es auch Erweiterungen, die genau dies tun. In der Sprache Erlang [3] ist dies sogar in den Sprachstandard eingeflossen. Abbildung 2 zeigt, wie sich das Beispiel aus Abbildung 1 in Python mit Hilfe einer Bibliothek vereinfachen lässt. Damit ist das Problem 1 von oben gelöst. Problem 2 bleibt weiter bestehen und Probleme 3 und 4 sind entschärft, aber nicht gelöst.

```
union Float_t {
    uint32_t raw;
    float f;
    struct    {
        uint32_t m : 23;
        uint32_t e : 8;
        uint32_t s : 1;
    } parts;
};
```

Abbildung 3: Fließkommazahl mit C-Datentypen

3 Echte Binäre Datentypen

Da man in der Programmiersprache C direkten zugriff auf das Speicherlayout von Datenstrukturen hat, ist es dort ein häufig gegangener Weg, das Speicherlayout, direkt als Binärformat zu verwenden. Abbildung 3 zeigt, wie man auf diese Weise die einzelnen Komponenten einer Fließkommazahl beschreiben kann. Dabei wird ausgenutzt, dass der Compiler einzelne Komponenten in der Reihenfolge, in der sie aufgelistet werden, im Speicher anordnet. Das genaue Layout ist jedoch bei weitem nicht allgemein spezifiziert und kann sich von Plattform zu Plattform, ja sogar von einer Compilerversion zur nächsten, ändern.

Allgemeiner ausgedrückt, wird ausgenutzt, dass jedem komplexen Datentypen eine Bitfolge implizit (durch das Speicherlayout) zugeordnet wird.

Binäre Datentypen sind nun die Schnittmenge von höheren Datenstrukturen mit Bitfolgen, d.h., dass ein Wert eines binären Datentyps gleichzeitig als Bitfolge und als ein Wert einer höheren Datenstruktur angesehen werden kann. Die Parsing- und Generierungsfunktionen sind also dem Datentyp inhärent.

An dieser Stelle muss noch einmal klar gestellt werden, dass die hier betrachtete Problemstellung der Umgang mit beliebigen Binärformaten ist. Systeme wie ASN.1 [5] ordnen zwar jeder Datenstruktur eine klar spezifizierte Bitfolge zu, doch nicht jedes Binärformat lässt sich mit ihnen darstellen.

Systeme, die binäre Datentypen als Grundlage zur Beschreibung von Bitformaten nutzen, sind BinPAC [6], Packet Types [2] und DataScript [1]. BinPAC und Packet Types sind vornehmlich zum Filtern von Netzwerkpaketen konzipiert. Dies spiegelt sich auch in ihrem Design wieder.

So kennt BinPAC keine Datentypen, die nicht ein vielfaches von acht Bit breit sind. Kleinere Komponenten müssen über dieselben umständlichen Ausdrücke extrahiert werden, wie es in Abschnitt 2 beschrieben wurde. Packet Types hat eine Restriktion, dass einzelne Komponenten Alignment-Grenzen nicht überschreiten dürfen. Diese Anforderungen spiegeln Optimierungen wieder, die für die Geschwindigkeit solcher Paketfilter notwendig sind. Dadurch werden allerdings auch viele Datenformat nur umständlich beschreibbar.

BinPAC besitzt außerdem die Möglichkeit C++-Code in die Datenformats-

pezifikation zu integrieren, welche beim Parsen ausgeführt wird. Dies ist ganz ähnlich zu semantischen Aktionen, wie sie in den meisten Parsergeneratoren verwendet werden. Verwendet man dieses Feature, verliert die Spezifikation allerdings ihre Allgemeinheit und kann nur noch mit anderem C++-Code benutzt werden.

Die Spezifikationssprache DataScript hingegen hat keine solchen Einschränkungen und die Spezifikationen sind vollkommen Plattform- und Sprachneutral. Solche Spezifikationssprachen sind im Prinzip formale Sprachen zur Beschreibung von Bitformaten und damit auch hervorragend zu deren Dokumention geeignet. Da sich aus den Spezifikationen Funktionen zum Parsen und Generieren der entsprechenden Bitfolgen erzeugen lässt und damit immer konsistent ist, wären auch Problem 2, 3 und 4 gelöst.

Um DataScript zu nutzen fehlen nur noch eine Reihe von Codegeneratoren für verschiedene Sprachen, die auch möglichst konfigurierbar sind. Leider liefert das DataScript-System hier keine Hilfestellung. Ein Codegenerator muss direkt auf dem DataScript-AST arbeiten, obwohl sich die Codegeneratoren viele Arbeitsschritte teilen. Das folgende Kapitel stellt ein Konzept vor, um dieses letzte Problem zu lösen.

4 Was kann partielle Auswertung für uns tun?

Unter partieller Auswertung versteht man im Prinzip eine aggressive Form von Konstantenfaltung (engl. „constant folding“). Zu einem gegebenem Programm und einem Teil seiner Eingabe wird ein neues Programm berechnet, dass sich genauso verhält, wie das Ausgangsprogramm nachdem es die Eingabe erhalten hat. Man sagt auch, dass das Ausgangsprogramm durch die Eingabe spezialisiert wird. Durch die fest gegebene Eingabe kann das Ausgangsprogramm unter Umständen sehr stark vereinfacht werden. Das kann soweit führen, dass die Spezialisierung und nachfolgende Ausführung des Restprogramms schneller sein kann, als die direkte Ausführung des Ausgangsprogramms.

Ein typisches Anwendungsbeispiel ist das Spezialisieren eines Interpreters mit einem Programm. Die Eingabe zu einem Interpreter ist der Quelltext des auszuführenden Programms und die Eingabe des Programms. Wenn man den Interpreter auf ein bestimmtes Programm spezialisiert, bleibt als Eingabe für das Restprogramm nur noch die Eingabe des auszuführenden Programms. Mit anderen Worten: das Programm wurde kompiliert. Dies wird als erste Futamura-Projektion bezeichnet.

Mit dieser Idee bieten sich verschiedene Anwendungsmöglichkeiten im Zusammenhang mit binären Datentypen an. Ausgangspunkt ist hier immer ein Programm, dass als Eingabe einen binären Datentyp und eine Bitfolge erhält und als Ausgabe die Daten als einen höheren Datentyp zurückliefert.

1. Die Parsing- und Generierungs-Funktionen werden mit dem binären Datentyp spezialisiert. Die Restprogramme ist nun Funktionen, die eine Bitfolge nach einem bestimmten Datentyp parsen bzw. die zugehörige Bitfolge

zu einem Wert eines bestimmten Typs generieren. Dies ist sicherlich der typische Anwendungsfall.

2. Es wird eine Funktion spezialisiert, die zunächst die Parsing-Funktion mit dem binären Datentyp aufruft und danach nur eine bestimmten Komponente des Ergebnisses zurück liefert. Die Spezialisierung sorgt dafür, dass nur die Operationen ausgeführt werden, die für das Parsen dieser einen Komponente notwendig sind. Dies wird ein vielen Fällen das Restprogramm stark vereinfachen. Zudem wird das Konstruieren einer komplexeren Datenstruktur gespart. Mit einer solchen Funktion kann man also direkt aus der Bitfolge lesen.

3. Es wird eine Funktion spezialisiert, die zunächst parst, eine Manipulation der Daten durchführt und das Ergebnis wieder in eine Bitfolge umwandelt. Auf diese Weise lassen sich sehr elegant effiziente Bitmanipulationen beschreiben.

4. Die Parsing- und Generierungs-Funktionen werden nicht spezialisiert. Dies ist notwendig, wenn der Datentyp zur Compile-Zeit nicht bekannt ist.

Diese Vorgehensweise hat einige Vorteile.

1. Da sich diese Varianten automatisch aus der Parsing- und Generierungs-Funktion abgeleitet werden, sind sie immer — die Korrektheit des Spezialisierers vorausgesetzt — zueinander konsistent.

2. Optimierungen im Layout des binären Datentyps, z.B. wie die Alignment-Anforderungen von Packet Types, können von dem Spezialisierer ausgenutzt werden. Dies ist schließlich genau eine der Hauptanwendungen von partieller Auswertung. Sind diese Optimierungen aber nicht vorhanden, kann der Algorithmus trotzdem arbeiten, wenn auch ineffizienter.

3. Es ist anzunehmen, dass eine Sprache die für partielle Auswertung optimiert ist, eine einfache Semantik besitzt und sich somit auch leicht in andere Sprachen übersetzen lässt. Des weiteren kann man die Maschinerie, die für die Spezialisierung benötigt wird, benutzen, um das Programm in eine Form zu bringen, die sich noch leichter in die Zielsprache übersetzen lässt.

5 Fazit und Offene Fragen

Binären Datentypen sind in der Entwicklung von Software eine fundamentales Konzept. Jedoch tragen Programmiersprachen dem keine Rechnung. Es gibt einige vielversprechende Ansätze, insbesondere DataScript. Wir haben die Möglichkeit vorgeschlagen, partielle Auswertung einzusetzen, um den Schritt der Codegenerierung flexibel und gleichzeitig effizient und wiederverwendbar zu gestalten.

Eine offene Frage ist die Nutzerfreundlichkeit eines solchen Systems. Um die Codegenerierung auf die jeweiligen Gegebenheiten anzupassen, wäre es von großem Vorteil, wenn der Nutzer Optimierungsregeln hinzufügen könnte. Die Korrektheit dieser Regeln vorrausgesetzt, stellt sich nun die Frage wie sich neue Regeln auf die Komplexität und das Terminierungsverhalten des Spezialisierungsprozesses auswirken. Es sollte möglich sein, weitere Regeln zu definieren, ohne das der Nutzer genaue Kenntnisse des partiellen Auswertesystems hat.

Ein von den bisher untersuchten Bitformatsprachen nicht beachtetet Thema ist die Interpretation der innersten Bitfelder. Häufig werden Bitformate, wie z.B. Fixkommazahlen, verwendet, die von den Programmiersprachen nicht nativ unterstützt werden. Es wäre sehr nützlich, wenn die Umwandlung und Rückumwandlung zwischen der Bitfolge und einer semantischen Entsprechung des Wertes formal in der Spezifikation des Bitformates geschehen könnte. Dabei besteht allerdings das Problem, dass die Bitrepräsentation und auch die semantische Repräsentation nicht immer eindeutig ist. Eine erste Idee besteht darin, diese Interpretation mit Hilfe von einer Art Pseudo-Bijektion zu beschreiben. Die Interprationsspezifikation für ein Feld könnte man mit Hilfe einer Kombination von aus grundlegenden Bausteinen beschreiben, ähnlich, wie in [7] reversible Parser aus partiellen Isomorphismen konstruiert wurden.

Literatur

[1] Godmar Back. DataScript - A specification and scripting language for binary data. *Generative Programming and Component Engineering*, pages 66–77, 2002. URL `http://link.springer.com/chapter/10.1007/3-540-45821-2_4`.

[2] Satish Chandra and Peter J. McCann. Packet Types. In *Workshop on Compilers Support for Systems Software*, Atlanta, 1999.

[3] Ericsson AB. Erlang/OTP System Documentation 5.10.2. Technical report. URL `http://www.erlang.org`.

[4] Scott Griffiths. Python Bitstring. URL `http://code.google.com/p/python-bitstring/`.

[5] ITU-T Study Group 17. ASN.1 Project. URL `http://www.itu.int/ITU-T/asn1/`.

[6] Ruoming Pang, Vern Paxson, Robin Sommer, and Larry Peterson. binpac: A yacc for writing application protocol parsers. *Proceedings of the 6th ACM . . .*, 2006. URL `http://dl.acm.org/citation.cfm?id=1177119`.

[7] Tillmann Rendel and Klaus Ostermann. Invertible syntax descriptions: unifying parsing and pretty printing. *ACM Sigplan Notices*, 2010. URL `http://dl.acm.org/citation.cfm?id=1863525`.

Functional Kleene Closures

Nikita Danilenko

Institut für Informatik, Christian-Albrechts-Universität zu Kiel
Christian-Albrechts-Platz 4, D-24118 Kiel
nda@informatik.uni-kiel.de

Zusammenfassung

The Kleene algorithm for computing the star-closure of a matrix over a Kleene algebra is well-known and easy to implement in an imperative language. When implementing the same algorithm in a functional context there are several possible implementations which depend on the functional abstraction. We discuss three implementations that focus on different aspects of the „functional core“ of the actual algorithm and compare the complexities of the results.

PAF: A portable assembly language based on Forth

M. Anton Ertl*
TU Wien

Abstract

A portable assembly language provides access to machine-level features like memory addresses, machine words, code addresses, and modulo arithmetics, like assembly language, but abstracts away differences between architectures like the assembly language syntax, instruction encoding, register set size, and addressing modes. Forth already satisfies a number of the characteristics of a portable assembly language, and is therefore a good basis. This paper presents PAF, a portable assembly language based on Forth, and specifically discusses language features that other portable assembly languages do not have, and their benefits; it also discusses the differences from Forth. The main innovations of PAF are: tags indicate the control flow for indirect branches and calls; and PAF has two kinds of calls and definitions: the ABI ones follow the platform's calling convention and are useful for interfacing to the outside world, while the PAF ones allow tail-call elimination and are useful for implementing general control structures.

1 Introduction

Traditionally compilers have produced the assembly language for the various target architectures, and interpreters were written in assembly language. The disadvantage of this approach is that it requires retargetting for every new architecture. As a result, many such compilers and interpreters target only one or few architectures, and ports to new architectures often take quite a while.[1]

Portable assembly languages promise to solve this problem: Compile to (or implement the interpreter in) the portable assembly language, and the compiler or interpreter will work on a variety of architectures without extra effort. Of course the portable assembly language implementation has to be targeted for these architectures, but that effort can be reused (and possibly the cost shared) by several compilers/interpreters.

In this paper we present a new portable assembly language, PAF (for "Portable Assembly Forth"). There have been a number of languages that been designed and/or used as portable assembly languages (Section 2), so why introduce a new one?

1.1 Contributions

An issue that a number of them have had is that they require the code to be organized in functions that follow the standard calling convention (ABI) of the platform, which usually prevents tail-call optimization. PAF provides ABI calls and definitions for interfacing with the rest of the world, but also PAF calls and definitions, which (unlike ABI calls) can be tail-call-optimized and can therefore be used as universal control flow primitives [Ste77] (see Section 3.9 and 3.10).

*Correspondence Address: Institut für Computersprachen, Technische Universität Wien, Argentinierstraße 8, A-1040 Wien, Austria; `anton@mips.complang.tuwien.ac.at`

[1] E.g., AMD64 CPUs became available in 2003; The lina interpreter for AMD64 became available in 2008, the iForth compiler became available for AMD64 in 2009 (and 32-bit releases were stopped at the same time), and other significant Forth compilers like SwiftForth, VFX, and bigForth still do not offer 64-bit support in 2013. By contrast, the Gforth interpreter which uses a portable assembly language, was available there right from the start (thanks to our portable assembly language being there from the start), and we verified that by building and testing Gforth on an AMD64 system in August 2003.

Another problem is that indirect branches and calls have a high cost, because the compiler has to assume that every branch/call can reach any entry point. PAF introduces tags to specify which branches/calls can reach which entry points (see Section 3.9 and Section 3.9).

The most significant difference between PAF and Forth is that PAF contains restrictions that ensure that the stack depth is always statically determinable, so stack items can be mapped to registers. It is interesting that these restrictions are relatively minor and don't affect much Forth code; it's also interesting to see an example of Forth code that is affected (see Section 5).

2 Previous Work

This section discusses existing portable assembly languages, their features and deficiencies and why we feel the need for a new one.

2.1 C

C and its dialects, like GNU C, have been used as a portable assembly language in many systems: It is the prevalent language for writing interpreters (e.g., Python, Ruby, Gforth) and run-time systems; C has also been used as target language for compilers: (e.g., the original C++ compiler cfront, and one of the code generation options of GHC).

However, the C standard specifies a large number of "undefined behaviours", including things that one expects to behave predictably in a portable assembly language, e.g., signed integer overflow. In earlier times this was not a problem, because the C compilers still did what the programmer intended. Unfortunately, a trend in recent years among C compiler writers has been to "optimize" programs in such a way that it miscompiles (as in "not what the programmer intended") code that earlier compiler versions used to compile as intended. While it is usually possible to find workarounds for such a problem, the next compiler version often produces new problems, and with all these workarounds the direct relation from language feature to machine feature is lost.

Another problem of C (and probably a reason why it is not used as often as compiler target language as for interpreters) is that its control flow is quite inflexible: Code is divided into C functions, that can be called and from which control flow can return; the only other way to change control flow across functions is `longjmp()`.

Varargs in combination with other language features has led to calling conventions where the caller is responsible for removing the arguments from the stack. This makes it impossible in to implement guaranteed tail-call optimization, which would be necessary to use C calls as a general control flow primitive [Ste77].

As a result, any control flow that does not fit the C model, such as unlimited tail calls, backtracking, coroutining, and even exceptions is hard to map to C efficiently.

2.2 LLVM

LLVM is an intermediate representation for compilers with several front ends, optimization passes and back ends [LA04].

Unfortunately, it shares many of the problems of C: In particular, you have to divide the code into functions that follow some calling convention, restricting the kind of control flow that is possible. To work around this problem, it is possible to add your own calling convention, but that is not easy.[2]

LLVM was also promised to be a useful intermediate representation for JIT compilers, but reportedly its code generation is too slow for most JIT compiler uses.

LLVM supports fewer targets than C. Given that it also seems to share many of the disadvantages of C, it does not appear to be an attractive portable assembly language to me, despite the buzz it has generated.

[2] Usenet message `<KYGdnTH8PMyMpM7MnZ2dnUVZ_j-dnZ2d@supernews.com>`

2.3 C--

C-- [JRR99] has been designed as portable assembly language. Many considerations went into its design, and it appears to be well-designed, if a little too complex for my taste, but the project appears to be stagnant as a general portable assembly language, and it seems to have become an internal component of GHC (called Cmm there).

While C-- does not appear to be an option as portable assembly language for use in practical projects at the moment, looking at its design for inspiration is a good idea.

2.4 Vcode and GNU Lightning

Vcode [Eng96] is a library that provides a low-level interface for generating native code quickly (10 executed instructions for generating one instruction) and portably. It was part of a research project and has not been released widely, but it inspired GNU Lightning, a production system.

The demands of extremely fast code generation mean that GNU Lightning cannot perform any register allocation on its own. Therefore the front end has to perform the register allocation. It also does not perform instruction selection; each Lightning instruction is translated to at least one native instruction.

GNU Lightning also divides the code into functions that follow the standard calling convention, and one can call functions according to the calling convention. However, it is also possible to implement your own calling conventions and other control flow, because the front end is in control of register allocation, but (from reading the manual) it is not clear if this can be integrated with the stack handling by GNU Lightning und if one can use the processor's call instruction for your own calling convention.

It is possible to use better code generation technology with the GNU Lightning interface, and also to provide ways to use the processor's call and return instructions for your own calling convention.

With these changes, wouldn't the GNU Lightning interface be the perfect portable assembly language? It would certainly satisfy the basic requirements of a portable assembly language, but as a replacement for a language like C, it misses conveniences like register allocation.

3 Portable Assembly Forth (PAF)

3.1 Goals

- Portability: Works on several different architectures
- Direct relation between language feature and machine feature, i.e., if you look at a piece of PAF code, you can predict what the machine code will look like.

  However, the relation between PAF and the machine is not as direct as for GNU Lightning: There is register allocation and instruction selection, there may be instruction scheduling, and code replication. Instruction selection and instruction scheduling make better code possible (at the cost of slower compilation); register allocation interacts with these phases, and leaving it to the clients would require duplicated work in the clients, as register allocation is not really language-specific.
- Capabilities of the (user-mode part of the) machine can be expressed in PAF. However, this goal is moderated by the needs of clients and by the portability goal. I.e., PAF will at first only have language features that compilers and interpreters are likely to need (features can be added when clients need them); and machine features of particular architectures that cannot be abstracted into a language feature that can be implemented reasonably on the intended target machines will not be supported, either.

3.2 Target machines

While a portable assembly language can abstract away some of the differences between architectures, there are differences that are too difficult to bridge, and would lead PAF too far away from the idea of a direct correspondence between language feature and architectural feature, so here we define the class of machines that we target with PAF:

PAF targets general-purpose computer architectures, i.e., the architectures that have been designed as compiler targets, such as AMD64, ARM, IA-32, IA-64, MIPS, PowerPC, SPARC.

Memory on the target machines is byte-addressed with a flat address space; e.g., DSPs with separate X and Y address spaces are not target machines. The target machines use modulo (wrap-around) arithmetics and and signed numbers are represented in 2s-complement representation.

The target machines have a uniform register set for integers and addresses (not, e.g., accumulators with different size than address registers), and possibly separate (but internally also uniform) floating point registers.

3.3 Forth and PAF

Forth's low-level features are quite close to assembly language; e.g., like in assembly language, neither the compiler nor the run-time system maintains a type system, and the language differentiates between different operations based on name, not based on type; e.g., Forth has `<` for signed comparison and `U<` for unsigned comparison of cells (machine words), just like MIPS has `slt` and `sltu`, and Alpha has `cmplt` and `cmpult`.

Therefore Forth is a good basis for a portable assembly language. However, there are features that are problematic in this context: In particular, in Forth the stack depth is not necessarily statically determined (unlike in the JVM), even though in nearly all Forth code the stack depth is actually statically determined (known to the programmer, but not always the Forth system). So we change these language features for PAF.

A number of higher-level features of Forth are beyond the goal of a portable assembly language, so PAF does not support them.

On the other hand, there are a few things that are missing in standard Forth that have to be added to PAF, such as words for accessing 16-bit quantities in memory.

3.4 Example

The following example shows two definitions written in PAF:

```
                    \  cmpl %edx,%eax
: max               \  jle L28
  2dup >? if        \  ret
    drop exit endif \ L28:
  nip exit ;        \  movl %edx,%eax
                    \  ret
abi:xx- printmax {: n1 n2 -- :}
  "max(%ld,%ld)=%ld\n\0" drop
  n1 n2 2dup max abi.printf.xxxx-
  exit ;

\ Call from C:
\ main() { printmax(3,5); return 0; }
```

The first, `max`, looks almost like conventional Forth code, and corresponding assembly language code for IA-32 is shown in comments to the right. `max` does not have a fixed calling convention; the PAF compiler can set a calling convention that is appropriate for `max` and its callers (e.g., it can be tail-called). Since `max` does not follow the platform's calling convention, it cannot be called from, e.g., C code.

The second definition, `printmax`, follows the standard ABI of the platform (as indicated by using an `abi:` defining word. The `xx-` in `abi:xx-` shows that `printmax` expects and consumes two cells from the data stack and 0 floats from the FP stack and produces 0 cells and 0 floats; a C prototype for this definition could be `void printmax(long, long)`. `Printmax` calls `max`, and the compiler can choose the calling interface between the call and max; it calls `printf` using the standard calling convention with the call `abi.printf.xxxx-`, where the `xxxx-` indicates that four cells are passed as integer/address parameters and the return value of printf is ignored.

Locals are used in `printmax` but can be used in every definition. `Exit`ing from the definitions is explicit.

3.5 Registers

Several language features correspond to real machine registers: Stack items, locals, and values.

Stack items (elements) are useful for relatively short-lived data and (unlike locals) can be used for passing arguments and return values. There is no stack pointer and memory area specific to the stack, it's just an abstraction used by the compiler. Stack manipulation words like `DUP` or `SWAP` just modify the data flow and there is no machine code that directly corresponds to them (indirect consequences may be, e.g., move instructions at control flow joins).

Locals live within a definition and are a convenience: Local variables of the source language can be mapped directly to PAF's locals without needing register allocation or stack management in the front end. If a source local needs to be distributed across several PAF definitions (e.g., because a control structure of the source language is mapped to a PAF (tail) call), the local can be defined in each of these definitions, and the contants are passed on the stack across calls; this is not as convenient as one might like, but seems to be a good compromise.

Values are global (thread-local) variables whose address cannot be taken, so they can be stored in registers.

If stack items and locals don't fit in the registers, they are stored in a stack that is not visible to PAF code; this stack stores items from the stack[3], locals, and return addresses, so this does not correspond to the memory representation of, e.g., the data stack.[4]

If values don't fit in the registers, they are stored in global/thread-local memory.

3.6 Memory

The words `c@ uw@ ul@ ( addr -- u )` load unsigned 8/16/32-bit values from memory, while `sc@ w@ l@ ( addr -- n )` load signed 8/16/32-bit values from memory; `@ (addr -- w)` loads a cell (32-bit or 64-bit, depending on the machine) from memory; `sf@ df@ ( addr -- r )` load 32/64-bit floating-point values from memory. `c! w! l! ! ( x addr -- )` and `sf! df! ( r addr -- )` store stack items to memory.

3.7 Arithmetics

The usual Forth words `+ - * negate and or invert lshift rshift` correspond to the arithmetic and logic instructions present in every machine. There are also additional words like `/ m* um* um/mod sm/rem` that correspond to instructions on some machines, and have to be synthesized from other instructions on other machines.

[3]More precisely, the data stack for cells and the floating-point stack for floating-point numbers, but that detail plays no further role in this paper.

[4]Some languages have local variables whose address can be taken; it may be a good idea to provide a way to store them in this stack eventually, but for now such variables have to be stored elsewhere. The interaction of such a feature with, e.g., tail calls has to be considered first.

3.8 Comparison

The words `=?` `<?` `u<?` `f=?` `f<?` etc. compare two stack items and return 0 for false and 1 for true. They correspond to the Forth words `=` `<` `u<` `f=` `f<` etc., with the difference that the Forth words return −1 (all-bits-set) for true. A number of machines have instructions that produce 0 or 1 (MIPS, Alpha, IA-32, AMD64), while for others it is as easy to produce 0 or 1 as to produce 0 or −1, so "0 or 1" is more in line with the goal of the direct relation to the machine feature. An implementation of a 0-or-−1 language like Forth would use a sequence like `<? negate` for which good code can be generated easily.[5]

3.9 Control flow

... inside definitions

The standard Forth words `begin again until ahead if then cs-roll` are available in PAF and are useful for building structured control flow, such as `if ... then ... elsif ... then ... else ... end`.

While one can construct any control flow with these words [Bad90], if you want to implement labels and gotos, it's easier to use labels and gotos. Therefore, PAF (unlike Forth) provides that, too: `L:`*name* defines a label and `goto:`*name* jumps to it.

PAF also supports indirect gotos: `'`*name*`/`*tag* produces the address of label name, and `goto/`*tag* jumps to it. The tag indicates which gotos can jump to which labels; a PAF program must not jump to a label address generated with a different tag. E.g., a C compiler targeting PAF could use a separate tag for each `switch` statement and the labels occuring there.

These tags are useful for register allocation. One can use different tags when taking the address of the same label several times, and this may result in different label addresses, with the code at each target address matched to the gotos that use that tag (i.e., several entry points for the same PAF label).

Whichever method of control flow you use, on a control flow join the statically determined stack depth has to be the same on all joining control flows. This ensures that the PAF compiler can always determine the stack depth and can map stack items to registers even across control flow. This is a restriction compared to Forth, but most Forth code conforms with this restriction. Breaking this rule can be detected and reported by the PAF compiler.

PAF Definitions and PAF calls

A definition where the compiler is free to determine the calling interface is defined in the classical Forth way:

```
: name ... exit ;
```

The end of the definition does not produce an implicit return (unlike Forth), so you have to return explicitly with `exit`.

You call such a definition by writing its name, i.e., the traditional Forth way. You can explicitly tail-call such a definition with `jump:`*name*; this can be written explicitly, in the spirit of having a portable assembly language. Optimizing implicit tail calls is not hard, so the PAF compiler may do it, too.

We can take the address of a definition with `'`*name*`:`*tag*, call it with `exec.`*tag* and tail-call it with `jump.`*tag*. The tags indicate which calls can call which definitions.

The stack effects of all definitions whose address is taken with the same tag have to be compatible. I.e., there must be one stack effect that describes all of them; e.g., `( x x -- x )` is a valid stack effect of both `+` and `drop` (although the minimal stack effect of `drop` is `( x -- )`), so `+` and `drop` have compatible stack effects.

[5] Conversely, one might also decide to have `<` etc. instead of `<?` in PAF, and let the compiler handle the mismatch to some machines, but that would be somewhat against the spirit of a portable assembly language.

The use of tags here has two purposes: It informs the PAF compiler about the control flow; and it also informs it about the stack effect of the indirect call (while a Forth compiler usually has to assume that EXECUTE can call anything, and have any stack effect).

3.10 ABI definitions and ABI calls

We need to specify the stack effect explicitly as signature of an ABI definition or call. The syntax for such a signature is `[xr]*-[xr]*`, where x indicates a cell (machine word/integer/address) argument, and r a floating-point argument; the letters before the `-` indicate parameters, and the letters afterwards the results. The division into `x` and `r` reflects the division into general-purpose registers and floating-point registers on real machines, and the role these registers play in calling conventions.

A definition conforming to the calling convention is defined with `abi:`*`sig name`*. *Sig* specifies the stack effect, and indicates the correspondence between ABI parameters and PAF stack items. This signature is not quite redundant, e.g., consider the difference between the following definitions:

```
abi:x-x id   exit ;
abi:-   noop exit ;
```

These definitions differ only in the signature, yet they behave differently: `id` returns its argument, `noop` doesn't, and with ABI calling conventions, there is usually a difference between these behaviours.

You can call to an ABI-conforming function with `abi.`*`name`*`.`*`sig`*, where name is the name of the function (which may be a PAF definition or a function written in a different language and dynamically or statically linked with the PAF program). The signature specifies how many and which types of stack items to pass to the called functions, and what type of return value (if any) to push on the stack.

Putting the signature on every call may be a bit repetetive for human programmers, but PAF is mainly intended as an intermediate language, and an advantage of this scheme is that different calls to the same function (e.g., `printf`) can have different stack effects.

You can take the address of an ABI funtion with `abi'`*`name`* and call it with `abi-exec.`*`sig`*. There are no tail calls to ABI functions, because we cannot guarantee that tail calls can be optimized in all calling conventions.

Unlike PAF definitions, for ABI functions there is no point in tagging these function addresses, because the call always uses the ABI calling convention (whereas the compiler is free to determine the calling interface for PAF calls). The signature in indirect ABI calls has the same significance as in direct ABI calls.

3.11 Discussion

Why have two kinds of definitions and two kinds of calls?

The PAF definitions and calls allow to implement various control structures such as backtracking through tail calls [Ste77]. They also allow the compiler to use flexible and possibly more efficient calling interfaces than the ABI calling convention.

On the other hand, the ABI counterparts allow interfacing with other languages and using dynamically or statically linked binary libraries, including callbacks, and using PAF to build such libraries (e.g., as plug-ins).

4 Non-Features

This section discusses various features that PAF does not have and why.

4.1 Garbage collection

A number of virtual machines, e.g., the Java VM, support garbage collection. However, this feature significantly restricts what can be done. In particular, the data representations are restricted, and one cannot implement "unmanaged" languages or use a different data representation for a garbage collected language (e.g., the Java VM representation is quite different from how most Prolog or Lisp systems represent their data).

Even C--, which is intended as a portable assembly language for garbage collected languages does not implement garbage collection itself, but leaves it to the higher-level language, because that leaves the full freedom on how to implement data and garbage collection to the higher-level language [JRR99].

4.2 Types

PAF does not perform type checking during compilation, nor at run-time; also, there is no overloading of several operations on the same operator based on types. This is consistent with the descent from Forth, and non-portable assembly languages have the same approach.

However, in C-- the compiler knows about data types and uses that knowledge for overloading resolution. The disadvantage of such approaches is that it complicates the C-- compiler without making life easier for the front end compiler, which has to know exactly anyway whether it wants to perform, say, signed or unsigned comparison.

One may wonder about the "absence" of some operations in PAF; e.g., there is `< U<`, but only `= + - *`. The reason is that, on the two's-complement machines that PAF targets, these operations are the same for signed and unsigned numbers.

4.3 Debugger

Quite a bit of effort in C-- is devoted to supporting the standard debugger. For now there are no plans to make such an effort for PAF. C became a successful portable assembly language even though it has very little debugger support for languages that use it as intermediate language.

4.4 SIMD

Supporting SIMD instruction set extensions such as SSE, AVX, AltiVec etc. is not planned, mainly because few higher-level languages need such features. They can be added later if there is demand.

5 PAF vs. Forth

Some of the differences between PAF and Forth provide new insights into certain language features, and we take a closer look at that in this section.

The most interesting difference between PAF and Forth is in dealing with the stacks: PAF has restrictions and features that allow the compiler to statically determine the stack depth.

As a consequence, in PAF there is no need to implement the stacks in memory, with a stack pointer for each stack (data stack and return stack for cells, floating-point stack for floating-point values). In contrast, Forth needs to have a separate memory area and stack pointer for each stack, and while stack items can be kept in registers for most of the code, there are some words (in particular, `EXECUTE`) and code patterns (unbalanced stack effects on control flow joins), that force stack items into memory and usually also force stack pointer updates.

This property of Forth is avoided in PAF by requiring balanced stack effects on control flow joins (see Section 3.9), and by replacing `EXECUTE` with `exec.`*tag* (see Section 3.9); all definition addresses returned for a particular tag are required to have compatible stack effects, so `exec.`*tag* has a statically determined stack effect.

The effect on programs is relatively small: most Forth code has balanced stack effects for control flow anyway, and most occurences of `'` and `execute` can be converted to their tagged

```
\ Forth
: selector ( offset -- )
  create ,
does> ( ... o -- ... )
  @ over @ + @ execute ;

1 cells selector foo
2 cells selector bar

\ PAF
: foo ( ... o -- ... )
  dup @ 1 cells + @ jump.foo ;
: bar ( ... o -- ... )
  dup @ 2 cells + @ jump.bar ;
```

Figure 1: Defining method selectors in Forth and in PAF (simplified)

variants, because programmers keep the stack depth statically determinable in order to keep the code understandable.

However, there are cases where the restrictions are not so easy to comply with. E.g., object-oriented packages in Forth use EXECUTE for words with arbitrary stack effects. Programs using these words have a statically determined stack effect, too, but it is only there at a higher level; e.g., if you use a separate tag (and a separate `exec.`*tag*) for each method selector, typical uses would comply with the restriction, but in most object-oriented packages there is only one `execute`.

Figure 1 shows code for this example: the Forth variant defines a defining word `selector`, and the selectors are then defined with this defining word; in contrast, the PAF variant defines the selectors directly (and pretty repetetively), each with its own tag.

If you want to define a defining word for method selectors like you usually do in Forth, the tag would have to be passed around as a define-time parameter between the involved defining words. This support for higher-level programming is not required inside PAF (there we leave such meta-programming to the higher-level language), but if we want to transfer the tag idea back to Forth, we would have to do such things.

6 Related work

We have discussed C, LLVM, C--, and Vcode/GNU Lightning in Section 2.

There are projects that are similar to PAF in using a restricted or modified form of a higher-level language as portable assembler:

- The Python system PyPy uses a restricted form of Python called RPython as low-level intermediate language [AACM07].
- Asm.js[6] is a subset of JavaScript that is so restricted that it can serve as portable assembly language.
- PreScheme is a low-level subset of Scheme used as intermediate language for implementing Scheme48 [KR94].

In all these cases the base language is much higher-level than Forth, and it is much more of a stretch to create a low-level subset than for Forth..

Machine Forth (which evolved into colorForth) is a simple variant of Forth created by Chuck Moore, the inventor of Forth. It closely corresponds to the instructions on his Forth CPUs, but

[6] `http://asmjs.org/`

he also wrote an implementation for IA-32 that creates native code. The IA-32 compiler is very simple, basically just expanding the words into short machine code sequences.[7] It does not map stack items beyond the top-of-stack to registers, yet the generated code is relatively compact; this reflects the fact that machine Forth is close to the machine, including IA-32.

7 Conclusion

PAF is a subset/dialect of Forth that is intended as a portable assembly language. The main contributions of PAF are:

- *Tags* to indicate which indirect branches can reach which labels and which indirect calls can call which definitions. This restricts register allocation far less than unrestricted indirect branches and calls; and it needs less effort, and produces better results than trying to achieve the same result through program analysis.
- The division of definitions and calls into those conforming to the ABI/calling convention of the machine, and others for which the compiler can use any calling interface (and different ones for different sets of callers and callees). In particular, this allows tail-call optimization (unlike ABI calling conventions), which in turn means that we can use the calls as a primitive for arbitrary control structures (e.g., coroutining).
- Restrictions (compared to Forth) on the use of stack items that make it possible to have a static relation between stack items and registers for all programs, and avoid the need for a separate stack pointer and memory area for each stack. This highlights which Forth features are expensive and where they are used.

References

[AACM07] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. RPython: a step towards reconciling dynamically and statically typed OO languages. In Pascal Costanza and Robert Hirschfeld, editors, *DLS*, pages 53–64. ACM, 2007.

[Bad90] Wil Baden. Virtual rheology. In *FORML'90 Proceedings*, 1990.

[Eng96] Dawson R. Engler. VCODE: A retargetable, extensible, very fast dynamic code generation system. In *SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 160–170, 1996.

[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.

[KR94] Richard A. Kelsey and Jonathan A. Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):315–335, 1994.

[LA04] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization (CGO)*, pages 75–88. IEEE Computer Society, 2004.

[Ste77] Guy Lewis Steele Jr. Debunking the "expensive procedure call" myth or procedure call implementations considered harmful or lambda: The ultimate goto. AI Memo 443, MIT AI Lab, October 1977.

[7] `http://www.colorforth.com/forth.html`

INFINITY — Eine erweiterbare Programmiersprache

Marcus Frenkel
Lehrgebiet Programmiersysteme
Fernuniversität in Hagen
D-58084 Hagen
marcus.frenkel@feu.de

Friedrich Steimann
Lehrgebiet Programmiersysteme
Fernuniversität in Hagen
D-58084 Hagen
steimann@acm.org

Abstract. INFINITY ist eine Programmiersprache, welche die Erweiterung der eigenen Syntax und Semantik direkt im Programmcode erlaubt. Neue Syntaxregeln können zusammen mit ihrer Semantik definiert und direkt genutzt werden, ohne dass ein zusätzlicher Schritt zur Übersetzter-Genierung notwendig wird. In diesem Paper werden die grundlegenden Konzepte von INFINITY beschrieben, insbesondere die Art und Weise der Definition neuer Syntax und deren Anreicherung mit statischer und dynamischer Semantik.

1 Einleitung

Die vergangenen Jahrzehnte brachten die Entwicklung einer Vielzahl von Programmiersprachen mit sich; wann immer die bestehenden Sprachen nicht geeignet scheinen, ein bestimmtes Problem auszudrücken, wenn ein neues Programmierkonzept umgesetzt werden soll, wenn in einer bestehenden Sprache gewünschte Sprachkonstrukte nicht vorhanden sind oder wenn schlicht Unzufriedenheit mit der Syntax einer Sprache besteht, ist die übliche Antwort darauf die Entwicklung einer neuen Sprache oder die Erweiterung einer bestehenden in einen neuen Dialekt. Zwar existieren viele Werkzeuge zur Unterstützung bei der Sprachentwicklung, und auch viele existierende Programmiersprachen bringen verschiedene Möglichkeiten zur Syntaxerweiterung mit sich (etwa Racket [1], Scheme [2] oder Scala [3]), jedoch beschränken sich diese Ansätze meist auf reine Programmtransformationen, bei denen neue Syntax auf eine bereits bestehende Basissprache gemappt wird. Komplett neue Semantik, welche in der Basissprache so nicht vorgesehen ist, lässt sich damit nur schwer umsetzen.

Die hier vorgestellte Programmiersprache INFINITY soll dieses Problem umgehen, indem sie in ihrem Kern frei von jedem Programmierparadigma und auch frei von einer zugrundeliegenden Basissprache ist; der Kern der Sprache besteht lediglich aus einer Basissyntax, welche zur Definition weiterer Syntax benutzt werden kann, sowie einigen wenigen, primitiven Semantik-Regeln, auf denen aufbauend sukzessive beliebige weitere Semantik unabhängig vom Sprachkern oder einem bestimmten Paradigma definiert werden kann. Dabei nutzt INFINITY aus der objektorientierten Programmierung entlehnte Mechanismen, um eine größtmögliche Wiederverwendung bei gleichzeitiger Erweiterbarkeit von Syntax und Semantik zu gewährleisten.

Im weiteren Verlauf wird in Abschnitt 2 die INFINITY-Sprache selbst mit ihren verschiedenen Möglichkeiten zur Definition von Syntax und Semantik vorgestellt; in Abschnitt 3 werden einige Details der Referenz-Implementierung des INFINITY-Compilers (im Folgenden Referenzcompiler genannt) vorgestellt, gefolgt von der Diskussion zukünftig noch zu beantwortender Fragen in Abschnitt 4. Abschnitt 5 gibt einen kurzen Überblick über den aktuellen Stand der Forschung, und Abschnitt 6 enthält schließlich noch eine Zusammenfassung der hier vorgestellten Ansätze.

2 Die Sprache INFINITY

Das wichtigste Merkmal von INFINITY ist die Möglichkeit, Erweiterungen von Syntax und Semantik direkt in den regulären Programmcode zu integrieren; neue Semantik muss dabei nicht zwingend auf Konstrukte des Sprachkerns zurückgeführt werden, sondern kann relativ unabhängig von diesem frei definiert werden. Ähnlich der Möglichkeit bekannter Programmiersprachen, Klassen, Methoden und Variablen direkt nach ihrer Deklaration im Programmtext zu nutzen, können neue Syntax-Konstrukte in INFINITY direkt in der gleichen Übersetzungseinheit verwendet werden. Neue Sprachelemente werden dabei ausschließlich durch bereits vorhandene Sprachelemente deklariert, welche entweder aus dem Sprachkern von INFINITY stammen oder, aufbauend auf diesem, bereits eingeführt wurden.

In diesem Abschnitt sollen die wichtigsten Merkmale der INFINITY-Sprache anhand kurzer Beispiele vorgestellt werden.

Syntax-Definitionen Neue Syntax wird in INFINITY durch die Definition eines neuen Nichtterminal-Symbols mit der zugehörigen Produktionsregel hinzugefügt, wobei eine an die EBNF [4] angelehnte Syntax verwendet wird. Das folgende Beispiel zeigt die Definition eines neuen Nichtterminal-Symbols class mit der zugehörigen Produktionsregel in INFINITY, welche die (zumindest syntaktische) Deklaration einer leeren Klasse zulässt. Die Annotation top level statement erlaubt, die soeben definierte Syntaxregel an jeder Stelle im Programmcode zu verwenden, an der ein top level statement erwartet wird (weitere Details zu dieser Notation folgen gleich).

```
rule <top level statement>:
    class = "class", id, "extends", id, "{", "}";
```

class ist hierbei der Name des Nichtterminal-Symbols, "class", "extends", "{" und "}" stellen direkt zu parsende Token dar und id ist eine durch INFINITY vordefinierte Regel, welche zum Parsen von regulären Identifiern genutzt wird. Die so definierte Syntaxregel kann direkt nach ihrer Deklaration innerhalb der gleichen Übersetzungseinheit etwa in der folgenden Form genutzt werden und wird durch den INFINITY-Compiler in einen passenden (Teil-)Syntaxbaum geparst:

```
class C extends Object {}
```

Labels Jeder Syntaxstruktur innerhalb einer Produktionsregel kann ein eigenes Label zugewiesen werden, durch welches die zu dieser Struktur gehörenden, geparsten Syntaxbaum-Knoten später während der Überprüfung der Semantik identifiziert werden können. Die Zuweisung eines Labels erfolgt in spitzen Klammern hinter einer Syntaxstruktur, etwa in der folgenden Form, welche die beiden Label name und super den entsprechenden Syntaxstrukturen zuordnet:

```
rule <top level statement>:
    class = "class", id<"name">,
            "extends", id<"super">,
            "{", "}";
```

Annotationen Neben den Labels können in INFINITY Syntaxstrukturen weitere Eigenschaften zugewiesen werden, mit welchen der Parsing-Prozess beeinflusst werden kann. So kann etwa für eine Wiederholungs-Struktur die minimale und maximale Anzahl an Wiederholungen sowie ein möglicherweise zu parsendes Trennzeichen definiert werden. Die folgenden Regel-Definitionen erweitern etwa das bestehende Beispiel um die Möglichkeit, innerhalb einer Klassendeklaration

auch Methoden zu definieren, wobei zur Beschreibung des Komma-Trennzeichens zwischen Methodenparametern die separator-Annotation genutzt wird; mit dem Label methods wird die Menge der Methoden der Klasse referenziert:

```
rule: method = id<"returnType">, id<"name">,
               "(",
               { id<"paramType">, id<"paramName"> }<separator=",">,
               ")", ";";

rule <top level statement>:
    class = "class", id<"name">,
            "extends", id<"super">,
            "{", { method<"method"> }<"methods"> "}";
```

Regel-Hierarchie und abstrakte Regeln Ähnlich zur Klassenhierarchie aus objektorientierten Programmiersprachen können die in INFINITY definierten Syntaxregeln in einer Regelhierarchie angeordnet werden, indem bei der Deklaration einer Regel ihre (durch ihr Nichtterminal-Symbol eindeutig identifizierte) Eltern-Regel spezifiziert wird (so etwa bei der Deklaration der class-Regel oben geschehen, wo top level statement als Eltern-Regel angegeben wurde). Die so deklarierten Sub-Regeln können beim Parsen dann an jeder Stelle im Programmcode gematcht werden, an welcher der Parser ihre Eltern-Regel erwartet.

Als weitere Analogie zur Objektorientierung lassen sich, ähnlich zu abstrakten Klassen, in INFINITY auch abstrakte Regeln definieren, die zwar über ein Nichtterminal-Symbol, aber über keine Produktionsregel verfügen. Abstrakte Regeln können dementsprechend nie direkt geparst werden, sondern dienen in Produktionsregeln als Platzhalter für ihre konkreten Sub-Regeln. Das bereits erwähnte top level statement ist so eine abstrakte, durch den INFINITY-Kern bereitgestellte Regel, welche als Einstiegspunkt für weitere Statements innerhalb einer Übersetzungseinheit dienen kann.

Der folgende Code erweitert das bisherige Beispiel um eine abstrakte Regel primary und drei davon abgeleitete Sub-Regeln variable, creation und cast, welche wiederum als Rückgabewert-Ausdrücke der erweiterten method-Regel benutzt werden können:

```
rule: primary;

rule <primary>:
    variable = id;

rule <primary>:
    creation = "new", type, "(", { primary }<separator=",">, ")";

rule <primary>:
    cast = "(", id, ")", primary;

rule: method = id<"returnType">, id<"name">,
               "(",
                   { id<"paramType">, id<"paramName"> }<separator=",">,
               ")",
               "{", "return", primary<"returnValue">, ";", "}";
```

Vererbung von Semantik Um definierten Syntax-Regeln eine Semantik zuzuordnen, stehen in INFINITY verschiedene Möglichkeiten zur Verfügung. Die einfachste ist dabei das Erben von Semantik bestehender Regeln, was ähnlich zum Erben von Methoden von Superklassen in objektorientierten Sprachen wie Java funktioniert. Ausgehend von einer (abstrakten) Regel mit bereits zugeordneter Semantik wird über das Definieren einer Sub-Regel-Beziehung die Semantik

der Eltern-Regel an die Sub-Regel vererbt; auf diesem Weg kann also bereits bestehende Funktionalität mit neuer Syntax versehen werden.

Obiges Beispiel kann demzufolge einfach mit Semantik angereicht werden. Ausgehend von der angenommenen Existenz dreier abstrakter Regeln variable declaration, return statement und member method mit der für sie üblichen Semantik, kann das Beispiel folgendermaßen erweitert werden. Die method-Regel wird um eine Eltern-Regel erweitert und es werden zwei neue Regeln variable decl und return eingeführt; alle drei Regeln erben ihre Semantik von den erwähnten abstrakten Regeln. Zur Bekanntmachung der vorhandenen Regeln in der aktuellen Übersetzungseinheit ist zudem ein Import der Regeln nötig:

```
import variable declaration;
import return statement;
import member method;


rule <variable declaration>:
    variable decl = id<"type">, id<"name">;

rule <return statement>:
    return = "return", primary<"value">;

rule <member method>:
    method = id<"returnType">, id<"name">,
             "(",
                 { variable decl<"param"> }<"params", separator=",">,
             ")",
             "{", return<"body">, ";", "}";
```

Die Vererbung von Semantik wird auch genutzt, um INFINITY zu bootstrappen, indem ausgehend von einigen durch den INFINITY-Kern bereitgestellten abstrakten Regeln mit primitiver Semantik neue konkrete Syntaxregeln abgeleitet werden. Die somit definierte Basissprache kann wiederum zur Definition neuer Semantik genutzt werden (wie im Folgenden erklärt).

Hinzufügen von Semantik Neben dem Erben von Semantik ist es in INFINITY auch möglich, eine Regel mit komplett neuer statischer und dynamischer Semantik zu verknüpfen. Zu diesem Zweck wird in INFINITY das Konzept der Passes genutzt, welche sequentiell auf die einzelnen Knoten des Syntaxbaumes angewendet werden [5]. Jede Regel kann dabei eine Implementierung für jeden Pass bereitstellen, welche immer dann ausgeführt wird, wenn der entsprechende Pass auf Knoten angewendet wird, welche durch die Regel erstellt wurden. Die Pass-Implementierungen dienen zur Definition von statischer und dynamischer Semantik und müssen in einer Sprache geschrieben werden, die bereits in INFINITY implementiert wurde (etwa in einer rein durch Erben der Semantik von abstrakten Regeln des Sprachkerns definierten Basissprache).

Der INFINITY-Kern definiert bereits einige Standard-Passes, etwa zum Sammeln von Typen und Funktionen; weitere Passes können durch den Programmierer definiert werden, wobei die Ausführungsreihenfolge der Passes durch Constraints der Form $pass_1 < pass_2$ (in der Bedeutung, dass $pass_1$ vor $pass_2$ ausgeführt werden soll) angegeben wird.

Das bereits bestehende Beispiel kann so um weitere, statische Semantik erweitert werden, indem für die class-Regel eine Implementierung des vordefinierten Passes collect types in einer angenommenen, Java-ähnlichen Basissprache bereitgestellt wird; innerhalb dieser Implementierung wird ein neuer Typ vom passenden Namen bekannt gemacht, unter der Voraussetzung, dass er noch nicht existiert. Die Variablen this und parse sind hierbei implizit durch den Pass gegeben, wobei this den durch die Regel geparsten Knoten referenziert, auf welchen der Pass angewendet wird, und parse Zugriff auf den eigentlichen INFINITY-Compiler

gewährt, welcher wiederum verschiedene unterstützende Methoden bereitstellt. Die Felder name und super des this-Parameters verweisen auf die Kind-Knoten, die entsprechend der Produktions-Regel geparst wurden.

```
rule <top level statement>:
    class = "class", id<"name">,
            "extends", id<"super">,
            "{", { method<"method"> }<"methods"> "}";

    implement pass: collect types = {
        Type type = parse.getTypeByName (this.name);
        Type superType = parse.getType(this.super);

        if (type != null)
               parse.emitError("Type already defined:" + this.name);
           else if (superType == null)
               parse.emitError("Unknown super type:" + this.super);
           else
               type = parse.addType(this.name, superType);
        }
```

Neben der statischen Semantik ist ebenso die Definition neuer dynamischer Semantik möglich. Zu diesem Zweck kann auf den im INFINITY-Compiler integrierten Bytecode-Generator zurückgegriffen werden, welcher (im Referenzcompiler auf Basis der LLVM, siehe Abschnitt 3.3) Low-Level-Bytecode erzeugt. Das folgende Beispiel demonstriert die Erzeugung der dynamischen Semantik für eine Cast-Operation, indem mittels der durch den Compiler bereitgestellten Methode BuildBitCast der Bytecode für einen einfachen Cast erstellt wird. Die im Beispiel gezeigte Methode generate ist, ähnlich zu den Instanzmethoden der objektorientierten Programmierung, eine der Regel zugeordnete Methode, welche auf den durch die Regel erzeugten Syntaxbaum-Knoten aufgerufen werden kann. Dementsprechend wird generate durch eine Methode oder eine Pass-Implementierung eines weiter oben im Baums befindlichen Knotens aufgerufen; sie ruft wieder auch die generate-Methode eines untergeordneten Knotens (hier des Expression-Knotens) für weitere Code-Generierung auf.

```
rule <primary>:
    cast = "(", id<"type">, ")", primary<"expr">;
    ...
    Value generate(Parse parse) {
        Type targetType = parse.getTypeByName(this.type);

        Value exprValue = this.expr.generate(parse);
        return parse.BuildBitCast(exprValue, targetType);
    }
```

Besonders hervorzuheben ist hier, dass die zur Definition der Semantik genutzte, oben erwähnte Java-ähnliche Basissprache in einem vorherigen Schritt ebenfalls erst mit Mitteln der Semantik-Generation (Vererbung oder Hinzufügen) definiert wurde. INFINITY erlaubt somit, dass neu erstellte Syntax-Regeln mit ihrer verknüpften Semantik dazu genutzt werden können, weitere Semantik zu definieren; somit kann sukzessive aus einigen primitiven Basisregeln eine umfangreiche Sprache zur Semantikdefinition aufgebaut werden.

Featherweight Java Die bisher im Beispiel definierten Syntax-Regeln mit ihrer Semantik stellen im Übrigen bereits einen nicht unerheblichen Teil der Programmiersprache Featherweight Java dar [6]. Die komplette Syntax von Featherweight Java (unter Auslassung der Semantik) ließe sich in Infinity folgendermaßen notieren:

```
rule: type = id;
rule: primary;
rule: selector;
rule: term = primary, { selector };
rule <primary>:
    variable = id;
rule <primary>:
    creation = "new", type, "(", { term }<separator=",">, ")";
rule <primary>:
    cast = "(", id, ")", term;
rule <selector>:
    field access = ".", id;
rule <selector>:
    method invocation = ".", id, "(", { term }<separator=",">, ")";
rule: field = type, id;
rule: method = id, id,
               "(", { field }<separator=",">, ")",
               "{", "return", term, ";", "}";
rule: constructor = id,
                    "(",
                        { field }<separator=",">,
                    ")", "{",
                       "super", "(", { id }<separator=",">, ")", ";",
                       { "this", ".", id, "=", id, ";" },
                    "}";
rule <top level statement>:
    class = "class", id, "extends", id,
            "{",
               { field, ";" },
               constructor,
               { method },
            "}";
```

Mit Hilfe dieser Regeln kann durch den INFINITY-Parser beispielsweise das folgende Featherweight Java-Programm geparst und in einen Syntaxbaum umgewandelt werden:

```
class C extends Object {
    Object obj;

    C(Object obj) {
        super();
        this.obj = obj;
    }

    Object get(Object o) { return new C(o); }
}
```

Der Code kann, entsprechend der Definition der class-Regel, an jeder Stelle in einem INFINITY-Programm auftreten, an welcher ein top level statement erwartet wird (also etwa auf oberster Ebene einer Übersetzungseinheit), allerdings erst nach der Definition der entsprechenden Featherweight Java-Regeln (oder deren Import in einer anderen Übersetzungseinheit).

3 Implementierung

Wie der vorherige Abschnitt gezeigt hat, sind die wichtigsten Merkmale von INFINITY die Möglichkeit zur Definition neuer Syntaxregeln sowie die Zuordnung von statischer und dynamischer Semantik zu den Regeln. Neue Semantik muss dabei nicht zwingend auf bereits bestehende Semantik zurückgeführt werden, sondern kann mit Hilfe des INFINITY zugrunde liegenden Bytecode-Generators auf (quasi) Assembler-Ebene frei definiert werden.

3.1 Infinity-Kern und Bootstrapping

Der Kern von INFINITY besteht lediglich aus einigen Syntax-Regeln zum Definieren neuer Syntax sowie einigen abstrakten Regeln mit zugeordneter Semantik (welche im Compiler selber fest einprogrammiert ist – derzeit ungefähr 20 Regeln). Diese Basissyntax und die abstrakten Regeln können dazu genutzt werden, um zuerst eine einfache Basissprache und davon ausgehend weitere Programmiersprachen und Spracherweiterungen zu definieren. Die zur Verfügung stehenden Syntaxregeln entsprechen dabei weitgehend den Regeln der EBNF-Syntax mit Erweiterungen zur Deklaration und Implementierung von Regeln, Passes und Regel-Methoden. Die abstrakten Regeln des Kerns decken allgemein benötigte Funktionalität, wie etwa Methodendeklarationen, Anweisungen für bedingte Verzweigungen und Rückgabewerte sowie Ausdrücke für Funktionsaufrufe und einfache Vergleichsoperationen ab.

3.2 Parsing von INFINITY

Die Referenz-Implementierung des INFINITY-Compilers basiert auf einem Recursive Descent Parser [7] mit erweiterbarer Regelbasis. Jede Produktionsregel wird nach dem Parsen in eine Baumstruktur transformiert und in einer Map mit dem zugehörigen Nichtterminal-Symbol als Schlüssel gespeichert. Beim Parsen einer Eingabe-Datei wird der Eingabestrom durch den Parser gegen die gespeicherten Produktionsregel-Bäume gematcht, wobei diese rekursiv entsprechend des nächsten erwarteten Nichtterminal-Symbols hinabgestiegen wird; matcht eine Produktionsregel einen Teil des Eingabe-Stroms, wird dazu der entsprechende Syntaxbaum erstellt. Sollten an einer Stelle mehrere Produktionsregeln gültig sei, so wird der Syntaxbaum für die spezifischste Regel erzeugt; die spezifischste Regel ist hierbei diejenige, welche die meisten Terminalstrings enthält. Im Falle mehrerer matchender Regeln mit gleicher Anzahl an Terminalstrings wird ein Fehler mit Hinweis auf eine nicht entscheidbare Regelanwendung erzeugt.

3.3 Codegenerierung

Im Referenz-Compiler für INFINITY wird zur Bytecode-Generierung auf die LLVM (Low Level Virtual Machine) Compiler Infrastructure [8] zurückgegriffen, welche ihr eigenes Bytecode-Format, die LLVM Intermediate Representation (LLVM IR), definiert und einen Bytecode-Generator, einen Interpreter und einen Compiler sowie Linker zur Erzeugung von ausführbarem Maschinencode aus der LLVM IR mit sich bringt. Die LLVM IR ist eine Bytecode-Repräsentation auf sehr niedrigem Level, wodurch eine entsprechend große Flexibilität bei der Beschreibung dynamischer Semantik ermöglicht wird.

Der LLVM Interpreter ist zudem im Referenz-Compiler integriert und wird genutzt, um die Pass-Implementierungen und Methoden der Regeln zur Überprüfung der statischen und Generierung der dynamischen Semantik auszuführen. Da dynamische Semantik wiederum durch LLVM IR-Code beschrieben wird, kann sie direkt nach ihrer Definition im Programmcode durch den Compiler auch genutzt werden, was die gleichzeitige Definition und Nutzung von Regeln innerhalb einer Übersetzungseinheit erlaubt.

4 Offene Punkte

Die bisherige Spezifikation und Referenz-Implementierung von INFINITY umfasst die Definition von Syntax sowie statischer und dynamischer Semantik. Ein nächster Schritt ist die Implementierung eines Systems zur Behandlung von Fehlern während des Parsens, welches sowohl passende Meldungen über Syntax-Fehler generieren kann als auch im Fehlerfall ein Weiterparsen erlaubt. Weiterhin muss INFINITY um die Möglichkeit der syntaktischen Beschreibung von Kommentaren erweitert werden; Kommentare können nicht als regulärer Bestandteil von Produktionsregeln definiert werden, da sie an (fast) beliebigen Stellen auftreten können, so dass sie einen gesonderten Mechanismus zur Deklaration benötigen.

Ein wichtiger noch zu implementierender Punkt ist die Möglichkeit, Mehrdeutigkeiten während des Parsens aufzulösen. Mögliche Ansätze hierfür wären, dass Regeln innerhalb einer Übersetzungseinheit mit höherer Priorität gegenüber anderen markiert werden können, dass der Wirkungsbereich einer Regel auf eine bestimmte Coderegion beschränkt werden kann oder dass eine Regel innerhalb einer Coderegion deaktiviert werden kann.

Daneben befinden sich aktuell noch sprachbegleitende Features in der Entwicklung, etwa eine Entwicklungsumgebung mit integrierter Syntax-Hervorhebung auf der Basis der in einer Übersetzungseinheit verfügbaren Regeln; ein Debugger, welcher die Überwachung sowohl des eigentlichen Parse-Vorgangs, der Pass-Abarbeitung als auch der Ausführung des generierten Codes erlaubt; und ein System für Code Completion, Refactoring und Quick Fixes zur weiteren Unterstützung der Programmierung.

5 Stand der Forschung

Die Beschäftigung mit erweiterbaren Programmiersprachen ist seit vielen Jahrzehnten ein Gegenstand der Forschung. Obschon zwischenzeitlich verschiedene auftretende Probleme, insbesondere in Bezug auf die Benutzbarkeit erweiterbarer Sprachen durch Programmierer, die Forschung stagnieren ließen [9], wurden seitdem zahlreiche neue Ansätze zur Spracherweiterung vorgeschlagen, welche durch neue Technologien die bekannten Probleme umgehen [10]. Dabei lassen sich unterschiedliche Ansätze der Erweiterung unterscheiden.

Die ursprünglichste und am weitesten verbreitete Form der Spracherweiterung ist die Makro-Expansion in ihren verschiedensten Formen. Ein bekanntes Beispiel einfachster Makros sind die aus C bekannten Makros des Präprozessors [11], die eine einfache Form der Textersetzung darstellen. Hierbei werden lediglich während des Parsens gefundene Identifier durch eine vorher vorgegebene Zeichenkette ersetzt, eventuell auch unter Beachtung übergebener Makro-Parameter. Obgleich die Makros eine einfache Form der Syntax-Modifikation erlauben, sind sie in ihren Ausdrucksmöglichkeiten und Möglichkeiten zur Fehlerbehandlung stark eingeschränkt.

Demgegenüber stehen die syntaktischen Makros, die nicht nur eine einfache Form der Textersetzung sind, sondern direkt auf dem abstrakten Syntaxbaum operieren und somit eine sicherere Form von Programmtransformation darstellen [12]. Bekannte Vertreter hierfür sind etwa der C++-Templatemechanismus [13], das Makro-System aus Scheme [2] oder das aus Racket [1]; ähnlich zu den einfachen Makros für Textersetzung bleiben aber auch syntaktische Makros – im Unterschied zum Ansatz von INFINITY – weitgehend darauf beschränkt, neue Syntax auf eine bereits bestehende Basissprache zu mappen.

Makro-Expansion ist eine Form der Codetransformation, bei welcher im Allgemeinen (neue) syntaktische Strukturen, entweder auf lexikalischer Ebene einzelner Nichtterminale oder auf syntaktischer Ebene ganzer Sub-Bäume, in gültigen Code einer bereits bestehenden Sprache

umgewandelt werden. Betrifft die Transformation ein ganzes Programm, welches von einer (neuen) Sprache in eine andere (bereits bestehende) übersetzt wird, wird das als Programmtransformation bezeichnet [14]. Programmtransformation ist ein üblicher Ansatz zur Erstellung von domänenspezifischen Sprachen und wird durch viele bekannte Programme, insbesondere sogenannte Language Workbenches [15], genutzt, darunter Xtext [16], MPS [17], Spoofax [18], metafront [19] und ASF+SDF [20]. Im Unterschied zu INFINITY sind die Ansätze, die sich der reinen (Programm-)Transformation bedienen, darauf beschränkt, neue Syntax in die Syntax einer bestehenden Sprache zu transformieren. Das Einführen neuer dynamischer Semantik, die so in den möglichen Zielsprachen nicht abbildbar ist, wird dadurch erschwert, wenn nicht gar unmöglich gemacht.

Weniger eine erweiterbare Sprache, als vielmehr ein erweiterbarer Compiler ist JastAdd [21], welcher nicht die Erweiterung einer Sprache in der Sprache selbst erlaubt, sondern die Erweiterung des Compilers um neue Module und damit Funktionalität. Spracherweiterungen werden hierbei also getrennt vom eigentlichen Programmcode umgesetzt. Spracherweiterungen, die separat vom eigentlichen Programmcode erfolgen, bringen das Problem mit sich, dass Mehrdeutigkeiten in der Grammatik entweder bereits bei der Erstellung der Grammatik entdeckt beziehungsweise vermieden oder während des Parsens – soweit möglich – automatisch aufgelöst werden müssen. Im Unterschied dazu erlaubt INFINITY die dynamische Anpassung des Wirkungsbereiches von Syntaxregeln, um Syntax-Mehrdeutigkeiten so direkt im Programmcode fallweise auflösen zu können.

6 Zusammenfassung

Trotz umfangreicher Werkzeugunterstützung ist die Erweiterung bestehender Sprachen und die Erstellung komplett neuer Sprachen nach wie vor ein mühsames Unterfangen. Die hier vorgestellte Programmiersprache INFINITY hat zum Ziel, Programmierer bei der Entwicklung von Spracherweiterungen und neuer (domänenspezifischer) Sprachen zu unterstützen. Zu diesem Zweck erlaubt INFINITY die Definition neuer Syntax und die Verknüpfung dieser mit Semantik direkt in regulärem Programmcode, ohne dafür einen Zwischenschritt zur Compiler-Generierung zu benötigen. Da INFINITY nicht auf einer bestimmten High-Level-Basissprache aufbaut, sondern die dynamische Semantik komplett auf assemblernahen Low-Level-Bytecode abgebildet wird, liegt der Sprache auch kein bestimmtes Programmierparadigma zugrunde – INFINITY ist somit sehr flexibel und auch exotischere Sprachen und Spracherweiterungen können erstellt werden. Weiterhin erleichtert INFINITY die Definition neuer Semantik, indem bereits in vorhergehenden Schritten definierte Semantik genutzt werden kann, um neue Syntax-Regeln mit zusätzlicher Semantik zu verknüpfen.

Verweise

[1] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt und M. Felleisen, „Languages as Libraries“, *PLDI (2011),* pp. 132-141.

[2] R. Kelsey, W. Clinger und J. Rees, „Revised(5) Report on the Algorithmic Language Scheme“, 1998.

[3] M. Odersky, L. Spoon und B. Venners, Programming in Scala, Artima, 2008.

[4] „ISO/IEC 14977:1996(E)“, [Online]. Available: http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip.

[5] R. Bornat, Understanding and Writing Compilers: A Do It Yourself Guide, Palgrave Macmillan, 1979.

[6] A. Igarashi, B. C. Pierce und P. Wadler, „Featherweight Java: A Minimal Core Calculus for Java and GJ“, *TOPLAS 23:3,* pp. 396-450, 2001.

[7] A. V. Aho, M. S. Lam, R. Sethi und J. D. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986.

[8] „LLVM“, [Online]. Available: http://llvm.org/.

[9] T. A. Standish, “Extensibility in programming language design”, *ACM SIGPLAN Notices,* pp. 18-21, 7 July 1975.

[10] W. G. V., „Extensible Programming for the 21st Century“, *Queue - Programming Languages 2:9,* pp. 48-57, 2004.

[11] B. W. Kernighan und D. M. Ritchie, The C Programming Language, Prentice Hall, 1978.

[12] C. Braband und M. I. Schwartzbach, „Growing Languages with Metamorphic Syntax Macros“, *PEPM (2002),* pp. 31-40.

[13] B. Stroustrup, The C++ Programming Language, Addison Wesley, 1997.

[14] E. Visser, „A Survey of Rewriting Strategies in Program Transformation Systems“, *ENTCS, vol. 57,* pp. 109-143, 2001.

[15] M. Fowler, Domain-Specific Languages, Addison Wesley, 2010.

[16] „Xtext“, [Online]. Available: http://www.eclipse.org/Xtext.

[17] „MPS“, [Online]. Available: http://www.jetbrains.com/mps/.

[18] „Spoofax“, [Online]. Available: http://strategoxt.org/Spoofax/WebHome.

[19] S. M. I. Brabrand Claus, „The metafront system: Safe and extensible parsing and transformation“, *Science of Computer Programming 68:1,* p. 2–20, 2007.

[20] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser und J. Visser, „The ASF+SDF meta-environment: a component-based language development environment“, *Proc. of Compiler Construction,* 2001.

[21] T. Ekman und G. Hedin, „The JastAdd Extensible Java Compiler“, *ACM SIGPLAN Notices 42:10,* pp. 1-18, 2007.

An Efficient Scalable Runtime System for S-Net Dataflow Component Coordination

Clemens Grelck and Bert Gijsbers
University of Amsterdam
{c.grelck,b.gijsbers}@uva.nl

Abstract

S-Net is a declarative component coordination language aimed at radically facilitating software engineering for modern parallel compute systems by near-complete separation of concerns between application (component) engineering and concurrency orchestration. S-Net builds on the concept of stream processing to structure networks of communicating asynchronous components implemented in a conventional (sequential) language. In this paper we present the design, implementation and evaluation of a new and innovative runtime system for S-Net streaming networks. The Front runtime system outperforms the existing implementations of S-Net by orders of magnitude for stress-test benchmarks, significantly reduces runtimes of fully-fledged parallel applications with compute-intensive components and achieves good scalability on our 48-core test system.

1 Introduction

The multi-core revolution has brought parallel programming from the niche of high performance computing right into the main stream. As a consequence, programmers who had never thought about parallel processing in their application domains and who received no particular training in these issues are suddenly and rather unexpectedly exposed to the pitfalls of parallel processing. Conventional parallel programming is considered notoriously difficult. One reason for this is that it intertwines two different aspects of program execution: algorithmic behaviour, i.e. what is to be computed, and organization of concurrent execution, i.e. how a computation is performed on multiple execution units, including the necessary problem decomposition, communication and synchronization requirements. S-Net [7, 11] is a declarative coordination language whose design thoroughly avoids the intertwining of computational and organizational aspects. S-Net achieves a near complete separation of the concern of writing sequential application building blocks (i.e. *application engineering)* from the concern of composing these building blocks to form a parallel application (i.e. *concurrency engineering).*

S-Net defines the coordination behaviour of networks of asynchronous, stateless components and their orderly interconnection via typed streams. We deliberately restrict S-Net to coordination aspects and leave the specification of the concrete operational behaviour of basic components, named *boxes*, to conventional programming languages. An S-Net box is connected to the outside world by two typed streams, a single input stream and a single output stream. The operational behaviour of a box is characterized by a stream transformer function that maps a single data item from the input stream to a (possibly empty) stream of data items on the output stream. In order to facilitate dynamic reconfiguration of networks, a box has no internal state, and any access to external state (e.g. file system, environment variables, etc.) is confined to using the streaming network. This allows us to cheaply migrate boxes between computing resources and even having individual boxes process multiple records concurrently. Boxes execute fully asynchronously: as soon as data is available on the input stream, a box may start computing and producing data on the output stream. S-Net effec-

tively implements a macro data flow model, *macro* because boxes do not normally represent basic operations but rather individually non-trivial computations.

Although we do not target high performance computing applications with S-Net in particular, any user expects that S-Net programs run reasonably efficiently on parallel commodity hardware. In this paper we propose a highly efficient and scalable runtime system for S-Net, named Front. Front implements dynamically evolving S-Net streaming networks on multi-core multi-processor systems. Whereas previous S-Net runtime systems [6] merely served as proofs of concept for the macro data flow approach as such, with Front we combine all our experience gathered in the mean time to design and implement a low-overhead, high-performance runtime system that can achieve performance levels competitive with more machine-oriented parallel programming alternatives.

2 S-Net in a Nutshell

S-Net promotes functions implemented in a standard programming language into asynchronously executed stream-processing components, coined *boxes*. Both imperative and declarative programming languages qualify as box implementation languages, but we require any box not to carry over any information between two consecutive activations on the streaming layer. Each box is connected to the rest of the network by two typed streams: one for input and one for output. Messages on these typed streams are organized as non-recursive records, i.e. sets of label-value pairs. The labels are subdivided into *fields* and *tags*. The fields are associated with values from the box language domain; they are entirely opaque to S-Net.

Operationally, a box is triggered by receiving a record on its input stream. As soon as that happens, the box applies its box function to the record. In the course of function execution the box may communicate records on its output stream. Once the execution of the box function has terminated, the box is ready to receive and to process the next record on the input stream. On the S-Net level a box is characterized by a *box signature*: a mapping from an input type to a disjunction of output types. For example, box foo ((a,<b>) ->(c) |(c,d,<e>)); declares a box foo that expects records with a field labeled a and a tag labeled b. The box responds with an unspecified number of records that either have just field c, or else fields c and d as well as tag e. The associated box function foo is supposed to be of arity two: the first argument is of type void* to qualify for any opaque data; the second argument is of type int as the joint interpretation of tag values by the coordination and the component layer.

It is a distinguishing feature of S-Net that it neither introduces streams as explicit objects nor that it defines network connectivity through explicit wiring. Instead, it uses algebraic formulae to describe streaming networks. The restriction of boxes to a single input and a single output stream (SISO) is essential for this. S-Net provides five network combinators. Any combinator preserves the SISO property: any network, regardless of its complexity, is a SISO entity in its own right.

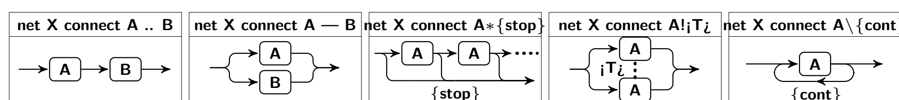


Figure 1: Illustration of the five S-Net network combinators, which are from left to right: serial composition, parallel composition, serial replication, parallel replication and feedback.

Let A and B denote two S-Net networks or boxes. Serial composition (A..B) constructs a new network where the output stream of A becomes the input stream of B, and the input stream of A and the output stream of B become the input and output streams of the combined network, respectively. Thus, A and B operate in a pipeline. Parallel composition (A|B) constructs a network where incoming records, depending on their type, are either sent to A or to B, and their output streams are merged to form the composed network's output stream.

In S-NET, the type system controls the flow of records. Each operand network is associated with a type signature inferred by the compiler. Any incoming record is directed towards the operand network whose input type is better matched by the type of the record.

The parallel and serial composition combinators have their infinite counterparts: serial and parallel replication combinators for a single operand network. The serial replication combinator A**type* constructs an infinite chain of replicas of A connected in series. The chain is tapped before every replica to extract records that match the type specified as the second operand. The parallel replication combinator A!<*tag*> also replicates network A infinitely, but this time the replicas are connected in parallel. All incoming records must carry the tag <tag>. This tag's value determines the network replica to which a record is sent. In addition to serial replication S-NET also features a more conventional feedback combinator A*type*. Here, records always enter subnetwork A. If an outgoing record matches the given type pattern, it is sent back to the entry point of A; otherwise, it leaves the compound network.

In addition to user-defined boxes S-NET features two primitive components: *filters* and *synchro-cells*. Filters mainly serve housekeeping tasks; they can split records, eliminate, duplicate and rename fields, and they support simple computations on (integer) tag values. All the same could be achieved by user-defined boxes, but it is engineering-wise more simple and execution-wise more efficient to have the ability to define such simple tasks on the coordination level as well. Synchro-cells are the one and only means to synchronise independent activities in S-NET. A synchro-cell, denoted as [| *type*, *type* |], allows us to merge records of the given types. A record that matches one of the type patterns is kept in the synchro-cell. As soon as a record that matches the other pattern arrives, the two records are merged into one, which is sent to the output stream. Incoming records that only match previously matched patterns are immediately forwarded. This bare metal semantics of synchro-cells captures the essential notion of synchronization in streaming networks. More complex synchronization behaviour, e.g. continuous synchronization of matching pairs in the input stream, can easily be expressed using synchro-cells and network combinators; details can be found in [5].

```
net Example ({Img} -> {Img})
{
  net Split connect [{R,G,B} -> {R} ; {G} ; {B}];
  net Sync connect [|{R},{G},{B}|] * {R,G,B};
  net Pipe connect (Split .. (fR|fG|fB) .. Sync .. Test) * {<done>};
}
connect Pre .. Pipe .. Post;
```

Figure 2: Example S-NET implementing a dynamic graphics filter pipeline

Fig. 2 shows a simple, yet non-trivial example S-NET coordination program that implements a dynamic graphics filter pipeline; a graphical illustration of the same program is sketched out in Fig. 3. The top-level pipeline consists of a preprocessing step (Pre) transforming an abstract image into its red, green and blue colour components, a dynamic filter pipeline (Pipe) and a postprocessing step (Post) that turns processed RGB image components back into the original image representation. The dynamically replicated filter pipeline, implemented with a star-combinator in Fig. 2 and shown in the central compound box in Fig. 3, consists of a splitter that divides an RGB-image (record) into three independent records carrying on the red, green and blue colour information, respectively. These records are routed to three custom filters by means of parallel composition. After the individual processing of colour components, separate red, green and blue records are captured and combined into a single record in the subsequent synchro-cell.

Since we assume a stream of images to be processed by our filter (pipeline) and a synchro-cell only synchronizes a single set of incoming records (see above), we embed the synchro-cell in another serial replication combinator. Consequently, the Sync network synchronizes and combines the first red value with the first green and the first blue value on the inbound

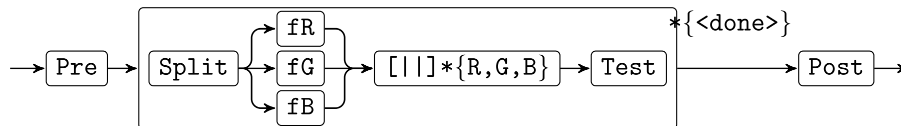


Figure 3: Illustration of the S-NET streaming network defined in Fig. 2

stream, the second red with the second green and blue value, and so on. On the combined RGB-image we run a simple test whether to continue filtering or to output the image. The decision is signaled by the presence or absence of the tag `<done>`, which is inspected by the star combinator and later removed in the postprocessing step.

To summarize, S-NET is an abstract notation to express concurrency in application programs in an abstract and intuitive way. It avoids the typical annoyances of machine-level concurrent programming. Instead, S-NET borrows the idea of streaming networks of asynchronous, stateless components, which segregates applications into their natural building blocks and exposes the data flow between them. However, S-NET is in no way confined to the area of streaming applications as several case studies successfully demonstrate [8, 13, 10].

3 Implementing S-Net

Fig. 4 illustrates the implementation architecture of S-NET. Going top to bottom, the S-NET compiler takes an S-NET coordination program and compiles it to the S-NET *Common Runtime Interface (CRI)*. This is a well-defined interface that exposes the structure of an S-NET streaming network as an application-specific call tree of application-agnostic library functions instantiated with again application-specific data structures. The library functions of the common runtime interface can be instantiated with alternative implementations and thus allow for entirely different technical realizations of S-NET.

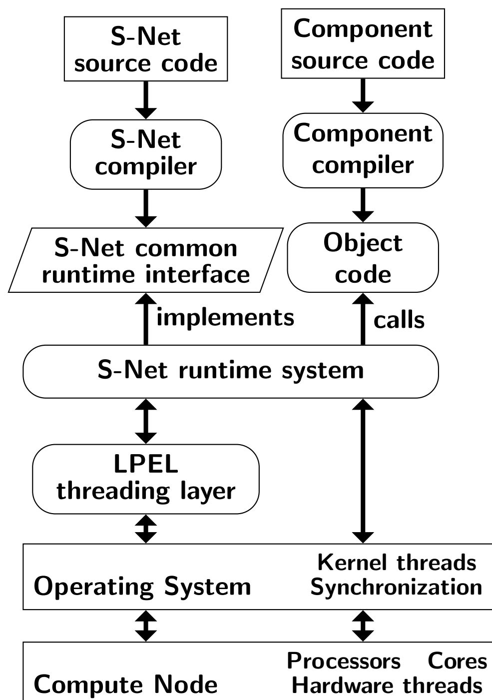


Figure 4: S-NET system architecture

The original S-NET runtime system [6], S-RTS, sticks closely to the intuition and the operational semantics of S-NET[12]. It sets up a system of communicating sequential processes (CSP). Each S-NET component is instantiated as a sequential process, which executes an event loop that reads a record from the input stream (potentially blocking on an empty stream) and processes that record. During component execution, one or more records may be emitted on the output stream. Termination of the box function completes the event loop and the component is ready to receive and process the next record on the input stream. On the S-NET language level stream split and merge points are implicitly represented in the semantics of the parallel composition combinator as well as both replication combinators. In the runtime system explicit *dispatch* and *collect* components take over these routing tasks. Unlike box components, a dispatcher has a single input and multiple output streams while a collector has multiple input streams and a single output stream. Both dispatchers and collectors implement the same event loop as boxes.

Both serial and parallel replication combinators describe conceptually unbounded streaming networks. The S-NET runtime system im-

plements them through demand-driven dynamic replication of networks; in Fig. 5 we illustrate this by a single level of instantiation and cloud symbols to represent future re-instantiations. It is the task of the dispatchers implementing serial and parallel replication combinators to identify the need for re-instantiation of subnetworks.

Assuming an unbounded (or at least fairly large) number of input records awaiting processing by an S-NET streaming network, we face the problem that choosing S-NET runtime components with non-empty input streams for execution without any further strategy may easily lead to a situation where many input records are loaded into the network, but very little progress is made towards completing the processing of individual records and emitting them on the overall output stream. Simultaneous in-memory representation of many records may exhaust the available memory. In order to ensure that an S-NET streaming network makes some progress towards completing records we use (fairly small) bounded buffers to implement streams. Accordingly, components may also block on full output buffers. This creates a form of *back pressure* that ensures sufficient progress in practice.

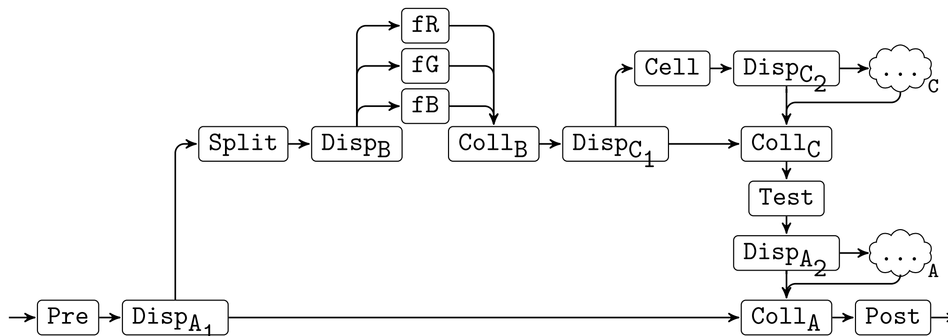


Figure 5: Illustration of the S-Net runtime system executing the streaming network defined in Fig. 2 and sketched out in Fig. 3; replication combinators are instantiated exactly once; clouds depict potential further instantiations

The S-NET runtime system itself is resource-agnostic. There are two alternative scenarios to map S-NET components to the underlying hardware for execution. One maps components one-to-one to kernel threads and leaves their scheduling to the operating system. We refer to this as S-RTS/PTH. The other scenario makes use of the tailor-made *Light-weight Parallel Execution Layer* (LPEL) [14] to explicitly map the dynamic number of S-NET components to a fixed, given number of kernel worker threads. LPEL serves several related purposes. It avoids the creation of a potentially large number of kernel threads and it uses efficient cooperative scheduling techniques. In the sequel we refer to this variant as S-RTS/LPEL.

4 The Front Work Stealing Runtime System

The design of the novel FRONT runtime system is based on an extensive body of experience with the original S-NET runtime system and the LPEL threading layer. S-RTS was mainly intended as a proof-of-concept for macro dataflow computing in the style of S-NET. The addition of LPEL was motivated by supporting meaningful profiling of S-NET streaming networks. With FRONT we now explicitly aim at scalable performance on large-scale (shared memory) compute servers.

4.1 Entity Graph vs Property Graph

Due to the presence of serial and parallel replication combinators in S-NET, the graph of communicating sequential processes is continuously evolving. It grows due to the demand-driven re-instantiation of argument networks, and it shrinks due to network garbage collection under certain circumstances [5]. Any change in the graph must be attributed to overhead

that competes for resources with productive box computations. In many situations, evolving the network graph additionally reduces the effectively exploitable concurrency. One of the key design ideas behind the FRONT runtime system is, thus, to replace the dynamically evolving graph of communicating sequential processes by two complementary graphs: the static *property graph* and the dynamically evolving *entity graph*.

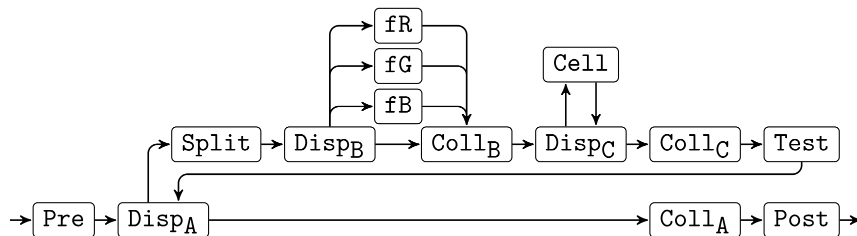


Figure 6: The static entity graph of the running example introduced in Fig. 2.

At program startup execution of the compiler-generated function call graph (the common runtime interface, see Section 3) creates the static *property graph*. In Fig. 6 we show the property graph for our running example. It merely contains placeholders for the replication combinators. The property graph contains all information required for running the network, e.g. types for routing decisions, and serves as a template for evolving the entity graph.

The entity graph is created strictly demand-driven. Initially, FRONT creates just the first entity, in our example this is `Pre`. Even the creation of the outgoing stream is postponed. The first record to leave `Pre` will detect the absence of an outgoing stream. The `Pre` entity structure contains a pointer to the `Pre` node in the static node graph of Fig. 6. This suffices to give the outgoing edge, which in turn gives the destination node. From the destination node the entity type is available (in this case a *star dispatcher*) together with application-specific type parameters (e.g. the termination pattern is `<done>`).

Why is it not possible to do with only a static component graph at runtime? The answer is simple: while boxes are stateless by design, whole networks may indeed be stateful as they may contain synchro-cells. At a synchro-cell it is important to match the right records according to the *semantics* of S-NET, which is based on actual replication.

4.2 Process-centric vs Data-centric View

The process-centric view characteristic for S-RTS implementing each entity by a dedicated thread of control, even if implemented as a logical, cooperative thread in LPEL, turned out to be expensive. Thus we aim at a radical change in the interpretation of streaming networks and abandon the process-oriented design. Instead, we create a fixed set of *worker threads* to be run on different cores at application startup. These workers roam the entity graph in search for work. When a worker finds an entity with a non-empty input stream, it locks the entity using a compare-and-swap (CAS) instruction while it processes that entity. One serious consequence of this design is that a thread can no longer block when the entity writes to an already full stream buffer. As the semantics of S-NET does not bound the number of records a box may emit, streams connecting entities must be unbounded, as well.

The question remains how workers find entities with non-empty input streams. Effectively roaming the entity graph would clearly be inefficient. Therefore, the unit of work scheduling in FRONT is the non-empty stream itself. Whenever a worker writes to a stream, it remembers this as a *license to read one record* from that stream for a future invocation at the destination entity. It stores this knowledge in a *stream reference structure*, which contains a pointer to the corresponding stream and a counter for the number of read-licenses it has for that stream. When new read-licenses arrive, the worker looks up the stream reference structure in a hash table which is indexed by a pointer to the stream. To exercise a read-license, the worker first attempts to lock the destination entity. If this succeeds, it decrements the number of read-licenses by one, reads one record from the stream and then invokes the entity. i

4.3 Execution Order

The way workers organize their collection of stream references determines the order in which they are processed. An entity graph can be regarded as a dynamic pipeline with parallel branches, which evolves from an input entity to an output entity. The edges in this graph are the streams which transport the records. In this picture we wish to preferably schedule those non-empty streams, which have the highest probability of quickly producing output. Each output record reduces the memory footprint and also provides the user with results.

Front stores stream references in a singly linked list per worker; the tail of the list is closer to the input entity and the head closer to the output entity. When a worker searches for a schedulable stream, it traverses this list from the head looking for a stream with a currently unused destination entity. When found, the worker locks the destination entity of the stream, reads one record from the stream and invokes the entity. If the invocation generates new output records which result in a new stream reference structure then these are inserted before the current position in the list. As a consequence, streams closer to the output entity are closer to the head of the list and, therefore, are prioritised.

4.4 Improved Data Locality

We further aim at avoiding the migration of records from core to core to improve data locality in the ubiquitous presence of hierarchical caches. We extend write operations to streams with a flag that identifies the last output record of some entity invocation. In this case the worker thread immediately continues with the follow-up entity and the current record, i.e. it follows the record through the entity graph. This data-centric (instead of entity-centric) solution has the added benefit that we save storing and retrieving read-licenses.

4.5 Input Control and Work Stealing

When the list of stream reference structures is empty a worker tries to obtain exclusive access to the input entity in order to retrieve records from the input parser. This strategy replaces the concept of back pressure through bounded streams in S-RTS to avoid overloading a streaming network with too many incoming records. Instead, new work is only admitted to the streaming network if workers are still idle.

If there is neither input on the global input stream or another worker has already locked the input entity, idle workers turn into thieve mode and examine the list of stream references of other workers. We store pointers to workers in a global array. Thieves iterate over this array when searching for work. They remember the previously visited victim and continue with the next worker (round-robin). To reduce contention with victims over their stream reference lists at most one thief at a time may visit a victim.

Where a worker looks first for more work when its own work list becomes empty is an important design decision. We choose workers to first check global input for more work before trying to steal work from other workers. This choice reduces the overhead created by many workers simultaneously aiming at stealing work that simply does not exist. Moreover, it helps to accelerate the initial ramp up phase of any S-Net network when the number of records in the system is still small and effectively no work exists that could be stolen. As only one worker at a time can lock and thus operate the input entity, new records may enter the streaming network while at the same time other workers aim at stealing work from their peers.

4.6 Concurrent Box Invocation

The S-Net language specifies box functions to be stateless. We can exploit this property to significantly increase concurrency by allowing multiple workers to invoke a box entity concurrently as soon as multiple input records are waiting in the input stream. For the purpose of experimentation, a per-box concurrency limit can be specified for now. We allocate for a

box entity an equivalent number of box contexts. Each box context has its dedicated outgoing stream, which ends at a shared per-box collector. The collector merges the incoming streams into one outgoing stream. When a worker invokes a box, it finds an unused box context, locks it and if the number of concurrent invocations is below the limit, it immediately unlocks the box entity, to allow for more concurrent invocations by other workers. The collector entity ensures that despite concurrent box invocations the stream order semantics of S-Net are preserved, i.e. records cannot coincidentally overtake other records.

The ability of the Front runtime system to invoke the same box instance multiple times concurrently if multiple input records are waiting to be processed is an important step to fully exploit the concurrency contained in an S-Net specification. Following the macro data flow approach the unit of computation in S-Net is the record, not the box component. Conversely, in a *communicating sequential process* implementation model, as the original S-Net runtime system does, opportunities for concurrent computations are regularly left out whenever multiple records start queuing in the input stream of a busy box component.

5 Performance Evaluation

We evaluate Front in comparison with the original S-Net runtime system with and without the LPEL threading layer. Our experimental system is a 48-core SMP machine with 4 AMD Opteron 6172 “Magny-Cours” processors running at 2.1 GHz and a total of 128 GB of DRAM. Each processor core has 64 KB of L1 cache for instructions, 64 KB of L1 cache for data, and 512 KB of L2 cache. Each group of 6 cores shares one L3 cache of 6 MB. The system runs Linux kernel 2.6.18 with Glibc 2.5. We first focus on fairly small benchmarks that stress test the runtime system design and implementation through large numbers of records, frequent expansion of dynamically replicated subnetworks and negligible computation within components before we tend towards real-world applications. Most sources are available online as part of the open source S-Net distribution [1]; more results can be found in [3].

5.1 Fibonacci Benchmark

With the Fibonacci benchmark we compare the performance of the runtime systems in creating and destroying entities and streams as well as the speed at which records are pushed through the entity graph. We do all computing with S-Net filters and minimize the influence of input parsing and output formatting by taking only one input record and producing a single output result, i.e. the argument and result of the Fibonacci function. Our implementation follows a divide-and-conquer approach such that all S-Net language constructs are used intensively. The number of created records is proportional to the value of the computed Fibonacci number. Fig. 7a shows that Front is about 50 times faster than S-RTS for this benchmark; Fig. 7b shows the (connected) rate at which records are created and destroyed.

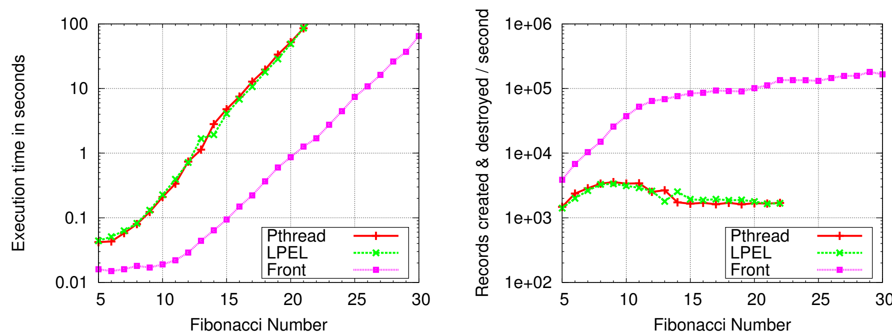


Figure 7: Fibonacci benchmark: **(a)** Execution time, **(b)** Record processing rate.

For FRONT this rate increases strongly up to $Fib(12)$ after which it increases weakly, whereas for S-RTS it diminishes between $Fib(11)$ and $Fib(15)$, regardless of the threading layer.

5.2 PowerOfTwo Benchmark

In S-RTS a collector entity faces the non-trivial problem of finding a non-empty stream among all incoming streams. FRONT solves this issue by only storing references to non-empty streams in the first place. We demonstrate the beneficial effect of this design choice with the PowerOfTwo benchmark, which emulates the recursive expansion of:

```
Po2 n = if n>0 then Po2 (n-1) + Po2 (n-2) else 1.
```

We use an index split combinator to create a large number of incoming connections to a collector entity. The maximum number of incoming connections is equal to half the value of the power of two of the input parameter. When we stepwise increase the input parameter we reach a point where the collector entity dominates performance for S-RTS, regardless of the threading layer. Fig. 8a shows execution times, Fig. 8b the corresponding record processing rates. Performance drops significantly for S-RTS when the number of incoming connections increases beyond 1,000, whereas scalability of FRONT is unaffected.

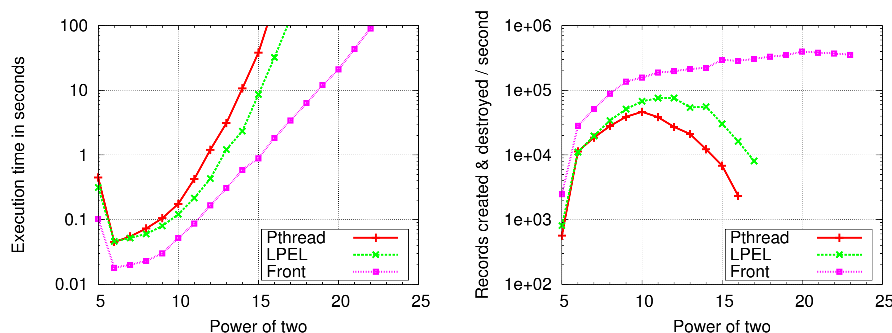


Figure 8: PowerOfTwo benchmark: **(a)** Execution time, **(b)** Record processing rate.

5.3 MTI-STAP Signal Processing

MTI-STAP is a signal processing application: Moving Target Indication using Space Time Adaptive Processing [9, 13]; it detects slow moving objects on the ground using an airborne radar antenna. We evaluate the performance of this application to see if concurrent box invocations in the runtime system can improve performance for existing S-NET applications.

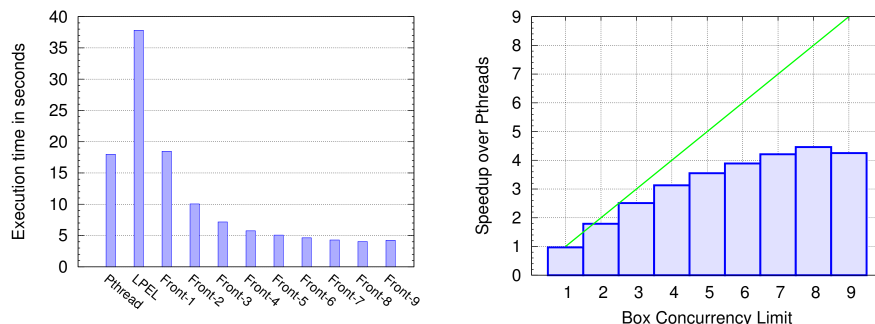


Figure 9: MTI-STAP: **(a)** Execution time, **(b)** Speedup for Concurrent Box Invocations.

Fig. 9a shows execution times for S-RTS/PTH, S-RTS/LPEL and FRONT. Here FRONT runs with the box concurrency limit set to numbers between 1 and 9 as indicated by the suffix:

The label FRONT-1 denotes the default configuration, i.e. no concurrent box invocations, while FRONT-9 denotes the configuration when up to nine workers may invoke a single box landing concurrently. Fig. 9b shows the speedup for increasing box concurrency limits relative to the execution time of S-RTS/PTH. This more clearly shows the performance gains by the concurrent box invocations. Of course performance gains by concurrent box invocations are highly application specific. Our implementation merely provides a mechanism for users to increase the exposed concurrency in their applications.

5.4 Cholesky Decomposition

Cholesky decomposition is a linear algebra problem: given a Hermitian positive-definite matrix A, find a lower triangular matrix L, such that $LL^T = A$, where L^T is the transpose of L. We use an implementation by Pāvels Zaičenkovs, University of Hertfordshire, based on the tiled algorithm proposed by Buttari et al [2]. We use this application to measure scalability and stepwise increase the number of cores. We incrementally add cores such that they share L3 caches and are part of the same processors and sockets as much as possible.

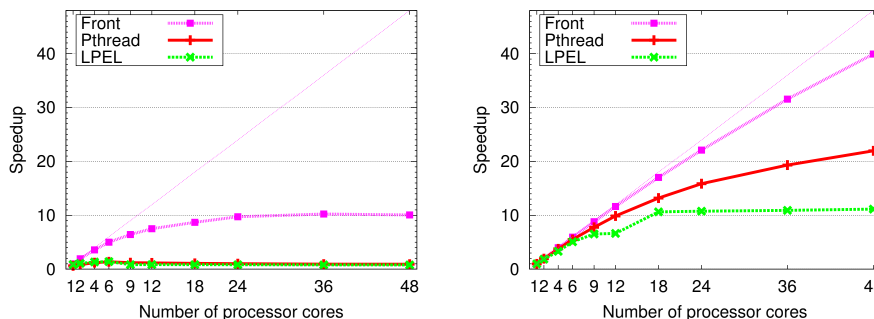


Figure 10: Speedup on Cholesky decomposition for two parameter sets: **(a)** Matrix size: 4096 by 4096, tile size: 64 by 64. **(b)** Matrix size: 8192 by 8192, tile size: 128 by 128.

Fig. 10a shows measurements 4096 by 4096 double precision floating point matrices using 64 by 64 tiles. This amounts to 32 KB per tile. Hence, two tiles fit into the L1 cache of 64 KB. The FRONT runtime system shows good speedups for 6 cores and diminishing speedups up to 24 cores. The S-RTS runtime shows just minor speedups up to 6 cores and no speedups beyond. In Fig. 10b we increase the amount of data per tile by four while keeping the total number of tiles identical. Hence, the number of S-NET entities remains the same as well; only the time spent in box components increases. Now, S-RTS/PTH also shows reasonable speedups, but less so S-RTS/LPEL. FRONT shows excellent speedup even for 48 cores. Increasing the number of cores from 36 to 48 still improves the performance by 21 percent, which we deem satisfactory considering that the algorithm also contains sequential sections.

6 Conclusions

We have presented the design, implementation and evaluation of the novel FRONT runtime system for the macro data flow coordination language S-NET. Aiming at highly efficient and scalable parallel execution FRONT dispenses with the intuitive interpretation of macro data flow coordination as a growing and shrinking system of sequential components communicating via bounded streams. Experimental evaluation shows that the FRONT runtime system outperforms the existing S-NET implementations by orders of magnitude for synthetic stress test benchmarks and considerably improves efficiency, resource utilization, throughput and scalability for real-world applications with compute-intensive box components.

Acknowledgements

This paper is an excerpt of a longer article presented at the 7th International Symposium on High-Level Parallel Programming (HLPP 2013) in Paris, France, to appear in the International Journal of Parallel Programming [4].

References

[1] S-Net system software distribution (2013). URL https://github.com/snetdev/snet-rts

[2] Buttari, A., Langou, J., Kurzak, J., Dongarra, J.: A class of parallel tiled linear algebra algorithms for multicore architectures. Parallel Comput. **35**(1), 38–53 (2009)

[3] Gijsbers, B.: An efficient scalable work-stealing runtime system for the s-net coordination language. Master's thesis, University of Amsterdam, Amsterdam, Netherlands (2013)

[4] Gijsbers, B., Grelck, C.: An efficient scalable runtime system for macro data flow processing using s-net. International Journal of Parallel Programming (2014). 7th International Symposium of High-Level Parallel Programming (HLPP'13), Paris, France, to appear

[5] Grelck, C.: The essence of synchronisation in asynchronous data flow. In: 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS'11), Anchorage, USA. IEEE Computer Society Press (2011)

[6] Grelck, C., Penczek, F.: Implementation Architecture and Multithreaded Runtime System of S-Net. In: S. Scholz, O. Chitil (eds.) Implementation and Application of Functional Languages, 20th International Symposium, IFL'08, Hatfield, UK, Revised Selected Papers, *Lecture Notes in Computer Science*, vol. 5836, pp. 60–79. Springer (2011)

[7] Grelck, C., Scholz, S., Shafarenko, A.: Asynchronous Stream Processing with S-Net. International Journal of Parallel Programming **38**(1), 38–67 (2010).

[8] Grelck, C., Scholz, S.B., Shafarenko, A.: Coordinating Data Parallel SAC Programs with S-Net. In: 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS'07), Long Beach, USA. IEEE Computer Society (2007)

[9] Le Chevalier, F., Maria, S.: Stap processing without noise-only reference: requirements and solutions. Radar, 2006. CIE '06. International Conference on pp. 1–4 (2006)

[10] Penczek, F., Cheng, W., Grelck, C., Kirner, R., Scheuermann, B., Shafarenko, A.: A data-flow based coordination approach to concurrent software engineering. In: 2nd Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM 2012), Minneapolis, USA. IEEE (2012)

[11] Grelck C., Shafarenko A. (eds.): Penczek, F., Grelck, C., Cai, H., Julku, J., Hölzenspies, P., Scholz, S., Shafarenko, A, Gijsbers B..: S-Net Language Report 2.1. (2013)

[12] Penczek, F., Grelck, C., Scholz, S.B.: An Operational Semantics for S-Net. In: B. Chapman, F. Desprez, et.al. (eds.) Parallel Computing: From Multicores and GPU's to Petascale, *Advances in Parallel Computing*, vol. 19, pp. 467–474. IOS Press (2010).

[13] Penczek, F., Herhut, S., Grelck, C., Scholz, S.B., Shafarenko, A., Barrière, R., Lenormand, E.: Parallel signal processing with S-Net. Procedia Computer Science **1**(1) (2010).

[14] Prokesch, D.: A light-weight parallel execution layer for shared-memory stream processing. Master's thesis, Technical University of Vienna, Vienna, Austria (2011)

An Efficient Native Function Interface for Java *

Matthias Grimmer, Manuel Rigger, Lukas Stadler,
Roland Schatz and Hanspeter Mössenböck

July 22, 2013

Abstract

We present an efficient and dynamic approach for calling native functions from within Java. Traditionally, programmers use the *Java Native Interface* (JNI) to call such functions. This paper introduces a new mechanism which we tailored specifically towards calling native functions from Java. We call it the *Graal Native Function Interface* (*GNFI*). It is faster than JNI in all relevant cases and more flexible because it avoids the JNI boiler-plate code.

GNFI enables the user to directly invoke native code from Java applications. We describe how *GNFI* creates call stubs for native functions that a just-in-time (JIT) compiler can optimize and how we embed these stubs into Java code. We introduce different approaches for calling native functions from within compiled and interpreted Java code. In particular, we describe how our approach embeds the call stubs into a Java application so that the JIT-compiled code consists of a direct call to a native function.

We evaluate the call overhead of *GNFI*. The measurements demonstrate a significant performance advantage of *GNFI* compared to JNI and the Java Native Access (JNA). Also, we evaluate our approach against JNI and JNA on a jblas matrix multiplication benchmark. The evaluation shows that *GNFI* outperforms JNI and JNA in compiled and interpreted mode by a factor of 1.9 in the best case.

1 Introduction

Programmers often have to call native code from Java. This is reasonable, e.g., when highly optimized native libraries exist or when an application has to use native legacy code. Other examples are language implementations on top of the JVM such as the Truffle language implementations [1, 2]. In these implementations, languages like C or Python have to call system libraries, because a reimplementation of these system libraries in Java would neither be feasible nor efficient.

Programmers typically call native functions using the *Java Native Interface (JNI)* [3]. JNI is a standardized interface which allows applications to switch execution from Java to a function available as native code and to manipulate Java objects from there. Programmers can pass arbitrary Java objects or primitives to such a function, which is declared *native* (but not implemented) on the Java side and implemented as a C function or C++ method. Native JNI functions have a specific name and signature which is implied by the JNI standard. Therefore, it is only possible to call native JNI functions from Java. It is not possible to call any other native function (e.g. a library function) from Java directly.

Whenever the Java application needs to call a native function it has to call a native JNI function (according to the JNI standard) which acts as a wrapper for the actual native target function.

Within this paper we define native code or native functions to be code that the machine can execute directly. For example, we consider both the compiled C or C++ JNI wrapper and the target function to be native code or a native function.

We define the native target function to be the actual function of interest. The native target function can follow any calling convention, such as the C calling convention. Hence, the JNI wrapper is not a native target function.

While it is common to write JNI wrapper functions to call native target functions, this has several disadvantages:

- For invoking a native target function two calls have to be performed. The first call is from the Java application to the JNI wrapper and the second from the JNI wrapper to the native target function. In principle, a single call would suffice, namely from the Java application to the native target function.
- JNI has an additional overhead when setting up parameters. Especially when JNI accesses Java arrays, no access method guarantees not to copy the array [4, 5] before using it on the native side.
- The call to a native method is opaque to the JIT compiler, meaning that it cannot inline the call. We refer to this as a *compilation barrier*. Because of the compilation barrier, the JIT compiler cannot optimize the native code of the JNI wrapper function that sets up the parameters.

*abbreviated version of `http://dx.doi.org/10.1145/2500828.2500832`

This three disadvantages are related to performance issues.

In this paper we present a native function interface for the Graal VM [6], called the Graal Native Function Interface (*GNFI*), which avoids these disadvantages.

We avoid the first disadvantage by applying a special inlining mechanism. Usually it is not possible to call native code directly from Java. To circumvent this issue, *GNFI* provides a Graal intermediate representation (IR) which we compile to code that assembles the parameters for the target. This Graal IR can then be inlined by the Graal compiler as if it were IR derived from Java bytecode. After inlining, we can remove the additional layer in most cases and can directly call the native target function.

The second disadvantage of JNI, i.e. the fact that JNI cannot guarantee obtaining references to Java arrays without copying, is also solved by *GNFI*. Since we use Graal IR to do native calls, we can pass pointers into the Java heap without the risk of possible garbage collections.

We avoid the third disadvantage by shifting the compilation barrier down to the call of the native target function. We do the conversion of parameters from Java types to native types on the Java side. This way, the JIT compiler can construct an intermediate representation of it and optimize this up to the native target function call. When using JNI, the parameter setup is hidden to the compiler because it happens on the native side, while we do it on the Java side.

GNFI offers only a small subset of the JNI functionality, namely to invoke native target functions, and it is specifically designed for this purpose. Therefore, the goal of *GNFI* is not to replace JNI, but to provide an alternative and faster way to dynamically invoke native target functions from within Java.

In terms of performance, *GNFI* is about 20% faster than JNI on calling native functions without parameters. *GNFI* outperforms the Java Native Access (JNA) interface by a factor of 10.3 for calls of parameter-less native functions. Using a modification of the jblas wrapper library [7], *GNFI* beats JNI and JNA by a factor of up to 1.9 on a matrix multiplication benchmark. Hence, we can demonstrate that *GNFI* has a significant performance advantage for native libraries when compared to JNI and JNA.

In terms of security and debuggability, *GNFI* raises the same issues as JNI. Using JNI, the programmer defeats Java's guarantees of safety and security [8].

The goal of *GNFI* is to improve performance by avoiding native wrapper functions and thus we neglect the issues of security and debuggability in this paper.

This report is structured as follows: Section 2 describes the relevant parts of the Graal VM and the HotSpot™ VM. In Section 3 we show how to use *GNFI* and introduce the concepts behind it. Then, we describe how we perform native calls and point out the different modes of execution. In Section 4, we describe the call stubs, which perform the conversion of calling conventions as well as the actual call to the native target function.

Section 5 describes how native calls are done from interpreted Java code and from compiled code. An evaluation of *GNFI* against JNI and JNA in Section 6 demonstrates the performance advantages of *GNFI* on micro benchmarks and a benchmark using the numerical library BLAS. The paper concludes with related work in Section 7 and the conclusion in Section 8.

2 System Overview

We implemented the Graal Native Function Interface in the context of the Graal OpenJDK project [6]. The Graal VM is a modification of the HotSpot™ VM. It uses Graal as a compiler, but all the other parts, including the interpreter, are from the HotSpot™ VM.

The HotSpot™ VM starts executing Java programs in the interpreter. If methods become hot, i.e., if their execution count exceeds a certain threshold, the VM compiles them using the Graal compiler. When a method has been compiled, Graal sends the resulting machine code back to the VM, which installs it into its data structures. We define *installed code* to be machine code where the VM makes sure that subsequent calls to that method immediately jump to the machine code, instead of using the interpreter to execute the bytecode of this method. Thus, a method can either be executed in compiled mode or in interpreted mode.

Graal parses Java bytecode into a form called the Graal Intermediate Representation (IR) [9, 10]. The Graal IR is a graph that captures the control flow and the data flow of instructions. It gets compiled to executable machine code by the Graal compiler. During compilation, the Graal compiler performs optimizations on this graph and installs the resulting machine code using the interface to the HotSpot™ VM. Besides code installation, this interface also allows the invocation of an installed method by providing a reference to it. Hence we can call any compiled Graal graph.

The Graal VM allows the installation of so-called *method intrinsics*. A method intrinsic is a pre-built graph that replaces a Java method's implementation.

The modifications necessary to implement the Graal Native Function Interface for the Graal VM are small: We extended the Graal IR with new nodes to execute calls to native functions. We also added a new capability to the public Graal API, namely the Native Function Interface. To improve the interpreter performance of *GNFI*, we extended the interface between the Graal compiler and the HotSpot^{TM} VM as described in Section 5.

3 The Graal Native Function Interface

Developers using Graal can easily use the Graal capabilities interface [11] to load the *GNFI* and access native target functions. In Section 3.1 we explain this procedure. Section 3.2 introduces two different modes for calling native target functions via *GNFI*.

3.1 Public Interface

A user can acquire and use the *NativeFunctionInterface* capability, as shown in the following example. We invoke the `floor` function (signature: `double floor(double value)`) of a math library written in C.

The usage of the *NativeFunctionInterface* follows a simple scheme as shown in Listing 1. Firstly, one has to load the capabilities for the *NativeFunctionInterface*, assuming the Graal VM supports it. The second statement uses the capabilities to load a math C library containing the native target function `floor`. Resolving the library results in a library handle which can be used to obtain a handle to the native target function, as shown in the third statement. To do so, the user has to provide the library handle, the name of the native target function and the signature. After these three steps, one can invoke the native target function, referenced by the *NativeFunctionHandle* arbitrarily often.

To invoke the native target function, the *NativeFunctionHandle* provides a method `call` which performs the native call.

In order to perform the call, *GNFI* has to convert the call on the *NativeFunctionHandle*, which is a Java method invocation, to a native call. The *NativeFunctionHandle* thus has to convert the calling convention from Java to C. For this calling convention conversion we first define the *GNFI Java calling convention* which all calls on *NativeFunctionHandle*s have to follow. This calling convention supports all parameter types that the native language requires.

```
NativeFunctionInterface ffi = Graal.
    getRequiredCapability(
    NativeFunctionInterface.class);
NativeLibraryHandle libraryHandle = ffi
    .getLibraryHandle("libMyMath.so");
NativeFunctionHandle functionHandle =
    ffi.getFunctionHandle(libraryHandle,
     "floor", double.class, double.class
    );
```

Listing 1: Obtaining a *GNFI* function handle

We encode all native types using boxed Java primitive values. In Java, primitive types do not inherit from `Object`. In order to treat them as objects, we have to use wrapper classes. We refer to instantiating such a wrapper class for a primitive type as boxing. For example, Java provides a class `Integer` which holds a value of the primitive Java type `int`.

The signature of the call method of a *NativeFunctionHandle* takes an `Object` array as a parameter. The values of this array are supposed to be boxed Java primitive values or one-dimensional Java arrays. Primitive native arguments take one slot in the array. Complex native arguments can take multiple slots. We call all native types that are atomic and can be encoded in a single Java primitive type *primitive native types*. *Complex native types* are native types that do not fit into a Java primitive type or are composite types. *Complex native types* are represented as a series of boxed Java primitive values.

Native calling conventions, such as the C calling convention, can require arguments to be passed by value or by reference. To pass arguments to a native target function by reference, it is necessary to know the memory address of the parameters. In Java this can be achieved by copying the data into a memory block according to the data layout of the native target language. The Java Unsafe API (available under restricted access in the OpenJDK) allows the allocation of memory on the native heap as well as loading data from it and storing data into it. The address of the allocated memory block can be stored into a Java `Long` value and passed to the native target function. An alternative is to pass the address of the data in the Java heap without copying it. This can only be applied for one-dimensional primitive Java arrays because their Java memory layout is the same as in C. To pass a Java array of a primitive type to the native target function, the user can directly pass the Java array object. The requirement to pass Java

array objects without copying, justifies the fact that the *GNFI Java calling convention* uses an object array for its arguments and boxes primitive types. To perform the call, *GNFI* takes the reference to the Java array object and determines the base address of the array data inside it. In Section 4 we explain why garbage collection is not a problem here. Depending on the application and purpose, the user can decide how to pass arrays by reference.

Native languages like C can also return complex types (e.g. structs) by value. In this case, the *GNFI Java calling convention* expects an additional parameter at position zero of the Java `Object` array. This parameter represents a pointer to a memory cell (encoded as a Java `Long` value) into which *GNFI* places the return value.

Listing 2 shows an example of how to use the *GNFI Java calling convention*. In this example we call the native target function `floor`. This call requires us to pass a `double` value to a native C function. We allocate an `Object` array with one slot, into which we put a Java `Double` value and pass the array to the native C function. The return value of the `call` is a Java `long` value which encodes a `double` value. The method `longBitsToDouble` can be used to convert the `long` value into a `double` value.

This example demonstrates that JNI and *GNFI* follow different approaches for handling parameters. *GNFI* abstains from implicit parameter conversions (i.e., from Java to C/C++) but rather requires the user to do the parameter preparation explicitly on the Java side. This removes the need for native JNI wrappers. However, it requires that the programmer aligns the data as expected by the native code. For example, the programmer is responsible for providing a memory block that represents a C structure before using a pointer to this structure as an argument. This arguments preparation on the Java side provides several advantages:

```
Object[] arg = new Object[1];
arg[0] =  new Double(1.5);
double result = Double.
    longBitsToDouble(functionHandle.
    call(arg));
```

Listing 2: Using the *GNFI* function handle to call a native target function

The first advantage in terms of performance is that we widen the compilation span of the JIT compiler. A JIT compiler, like Graal, benefits from this because the only compilation barrier is the call to the native target function itself.

The second advantage is that users do not have to maintain, compile and link C/C++ wrapper code when calling new native target functions. *GNFI* requires only the modification of Java code.

3.2 Modes of Execution

To convert the *GNFI Java calling convention* to a native calling convention we distinguish between two modes of the caller: the caller can either run in interpreted mode or in compiled mode. When it runs in interpreted mode we have to bridge two gaps on different levels (see Figure 1).

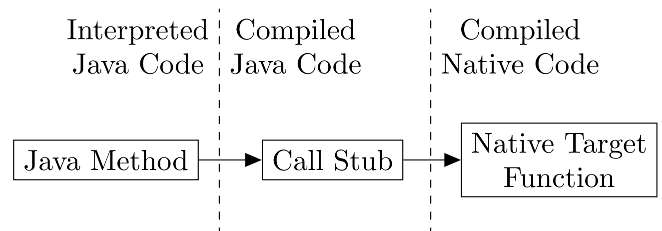


Figure 1: Layer gaps between the interpreted Java code and the native target function.

The first gap, on the left-hand side of Figure 1, is the call from interpreted Java code to a call stub, represented as installed code. For this paper we define a *call stub* to be an installed machine code method that converts the *GNFI Java calling convention* to the native calling convention and performs the actual call to the native target function. The call stub bridges the second gap on the right-hand side of Figure 1, and we describe it in more detail in Section 4. In order to call the machine code of the call stub from interpreted Java code, *GNFI* forwards the call to the HotSpot™ VM. The HotSpot™ VM bridges interpreted and compiled Java code and invokes the call stub code. The *NativeFunctionHandle* provides an interface for this installed call stub and uses the HotSpot™ VM facilities which is hidden from the users. Finally, one uses the NativeFunctionHandle to call the native target function following the *GNFI Java calling convention*.

If the calling Java method runs in compiled mode it can directly invoke the call stub because there is no need to bridge the gap between interpreted and compiled Java code.

4 The Call Stub

Whenever *GNFI* resolves a function handle, it is necessary to create a call stub. This call stub converts from the *GNFI Java calling convention* to the *native calling convention* (e.g., the C calling convention),

which requires the allocation of registers and the reservation of stack space according to the native calling convention.

Only machine code can do this allocation, hence it is not possible to implement the call stub method in Java. To overcome this issue, *GNFI* creates a Graal IR graph [9, 10] at run time and makes it the implementation of the call stub method. The signature of this Java method has an `Object` array as its argument and returns a `long` value.

To call such a stub method, one has to follow the *GNFI Java calling convention* as described in Section 3.

GNFI creates machine code by first synthesizing an IR graph and then compiling it into machine code using facilities of the Graal VM.

The IR graph of the call stub contains nodes that load the parameters from the passed `Object` array. Then, it unboxes these parameters or resolves their addresses (in case parameters are arrays). Finally, the call stub contains a special IR node that represents the native target function call with its parameters.

Since *GNFI* creates the IR for the call stubs, we can decide at which point garbage collection (GC) can happen in the Java VM. GCs can only happen when all threads running on the Java VM have reached a safepoint, i.e. a position where they can safely be stopped before GC is initiated.

The graph of the call stub does not contain any safepoint at which GC can happen, neither does the code of a native target function. Hence passing Java object references to native target functions is safe because the VM cannot do a GC while the native target function is executed. The fact that other Java threads have to wait for the native call to return when a GC is pending might be seen as a restriction, but it also comes with the advantage that native target functions can safely access Java objects. If the blocking of the GC during the execution of a native target function is not acceptable, one has to resort to JNI.

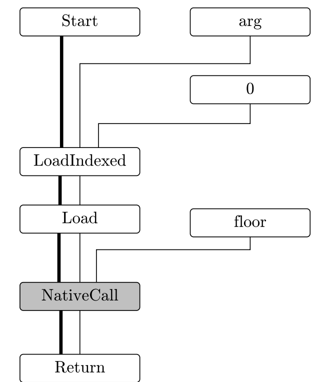


Figure 2: IR graph of the call stub for the native target function `floor`

Figure 2 shows the Graal graph of the call stub of our C function `floor`. In this graph, thin edges denote the data flow, while thick edges denote the control flow of the graph. The `arg` node is the parameter of this `callstub` method. It is the reference to the `Object` array that contains the parameters for the `floor` function. The *LoadIndexed* loads the first element of the argument array. The *Load* node loads the `double` value from the `Double` object and passes the value to the native call node. Beside the argument, the native call node also has a pointer to the function `floor` as an input.

The native call node is compiled to machine code that performs the platform-specific call to the native target function. The machine code stores the parameters into the right registers and stack slots according to the native calling convention and finally calls the native target function. The implementation, described in this paper, produces machine code according to the AMD64 Application Binary Interface (ABI). Of course this ABI can easily be exchanged for different platforms.

To install the machine code of this call stub in the VM we use the interface between the Graal compiler and the HotSpot^{TM} VM. Every subsequent call to a *NativeFunctionHandle* will invoke this installed machine code.

The call stub only depends on the signature of the native target function. This signature has to be specified by the programmer.

If the native target function has a fixed signature, i.e., it has a fixed number of argument types and not a variable number like the C function `printf`, the signature can be seen as a compile-time constant. Therefore, the call stub and its IR graph are also compile-time constants and we can create them statically.

As most target functions have a fixed signature, their call stubs are compile-time constants. A constant call stub has the advantage that *GNFI* can inline the call stub's IR graph into the Java application. Inlining reduces the call overhead to a minimum and created opportunities for further optimizations. In Section 5 we explain how we inline call stubs.

5 The Native Function Call

When an interpreted part of a Java application calls a native target function, there are two gaps that *GNFI* has to bridge. As Figure 1 shows, the first gap is between the interpreted Java application and the compiled Java call stub. The second gap is between the Java calling convention and the native calling

convention, which we described in Section 4.

Whenever the user of *GNFI* executes a *NativeFunctionHandle* by using the *GNFI Java calling convention* and the caller runs in interpreted mode, *GNFI* has to invoke the call stub. The call stub is represented by an installed code. To invoke it, the *GNFI* uses the interface between Graal and HotSpot^{TM} VM. This interface offers a method that tells the HotSpot^{TM} VM to execute an installed code, which is in our case the call stub.

```
Object[] arg = new Object[1];
arg[0] = new Double(1.5);
double c = 1.0 + Double.
    longBitsToDouble(functionHandle.
    call(arg));
```

Listing 3: Addition example using the native function `floor`

To speed up communication between Graal and HotSpot^{TM}, we extended the interface by an additional native declared execute method (Java side of the interface) which has a machine code implementation on the HotSpot^{TM}side (native side of the interface) that converts the Java interpreter calling convention to the Java compiled calling convention before invoking the actual call stub.

A *NativeFunctionHandle* can also be called from compiled code. When the Graal VM compiles the caller method we can make use of Graal *intrinsifications*. Instead of invoking the call stub by using the Graal to HotSpot^{TM}bridge (from interpreted Java code to compiled Java code), we intrinsify the call method of the *NativeFunctionHandle* object.

The intrinsification inserts a special IR node *FunctionHandleIntrinsificationNode* into the IR of the caller method. The *FunctionHandleIntrinsificationNode* replaces the call on the *NativeFunctionHandle*. To illustrate this intrinsification we use the native C function `floor` again. In Listing 3 we introduce a new example where we add 1 to the result of the native call to the `floor` function. The argument for this call is 1.5. To prepare the arguments array `arg` we box the value 1.5 and store it at position zero.

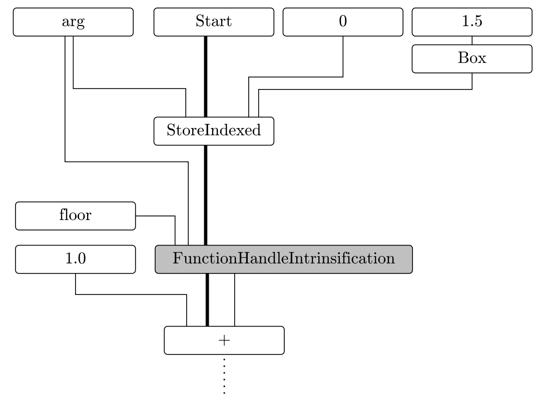


Figure 3: IR graph for Listing 3 containing a *FunctionHandleIntrinsificationNode*

Figure 3 shows the IR graph of this example. As we can see, the intrinsification of the *NativeFunctionHandle* replaces the call on its `call` method by a *FunctionHandleIntrinsificationNode*. The *StoreIndexed* node stores the boxed value 1.5 at position zero of the array.

The key observation in this graph is that it does not contain the call of the *NativeFunctionHandle* any more. The call is replaced by the *FunctionHandleIntrinsificationNode*. Finally the graph shows the addition of the result of `floor` and the value 1.

The *FunctionHandleIntrinsificationNode* exhibits two different behaviors, depending on the call stub it represents. Call stubs for variadic functions are run-time variables, because the number of arguments (and therefore the call stub) can vary. To call a variadic native target function it is necessary to invoke the call stub, resulting in two calls, one from the compiled code to the call stub and one from the call stub to the native target method. Call stubs for non-variadic functions are compile time constants, because the call stub does not change. If the call stub is a compile-time constant, we can reduce the number of calls to one.

In the first case, where the call stub is a run-time variable, the *FunctionHandleIntrinsificationNode* calls the installed code that represents the call stub. This means that the node replaces itself by a direct call to the installed call stub.

For this call it is not necessary to convert the arguments from the Java interpreter calling convention to the Java compiled calling convention. Instead, all arguments are already in place. Therefore, we can directly perform the call to the call stub.

With respect to our example of Listing 3 and assuming that the native target function `floor` is a run-time variable, this would mean that the *FunctionHandleIntrinsification* replaces itself by a direct call to the call stub.

In the second case, where the call stub and its graph are compile-time constants, we can directly inline this graph. Instead of a call to the installed code of the call stub, the *InstalledCodeIntrinsificationNode* replaces itself by the IR graph of the call stub. After compilation, the machine code has only one call site left, namely the site which directly calls the native target function. Thus, inlining the IR graph of the

call stub enables the Graal compiler to do optimizations on the whole range up to the native call itself.

In the example of Listing 3 the stub can be inlined because `floor` has a static signature. The left part of Figure 4 shows the graph that results from replacing the *InstalledCodeIntrinsificationNode* by the actual call stub graph. In this graph all call sites, except the native call to the `floor` function, are gone. After inlining the call stub, the Graal compiler is able to apply its optimizations up to the point where the call to the native function appears.

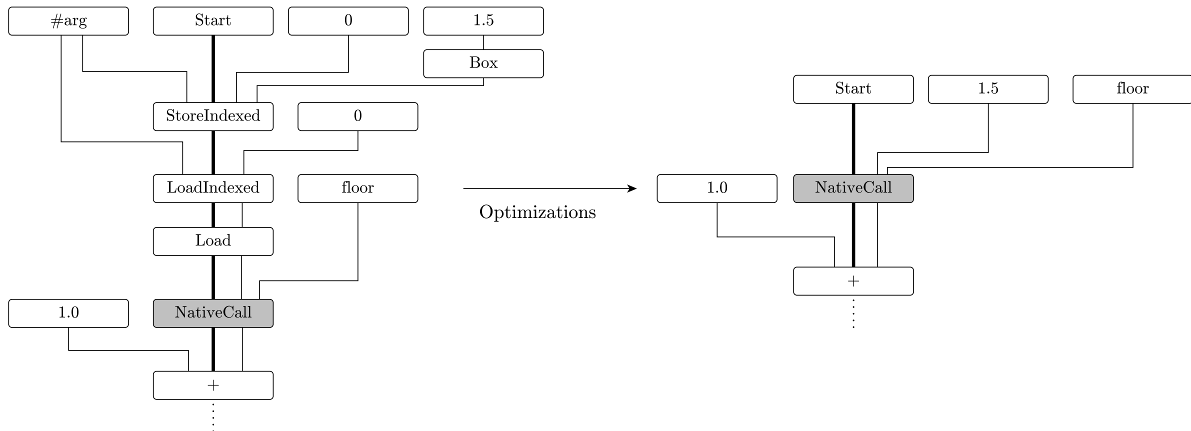


Figure 4: IR graph for Listing 3 with the call stub inlined (before and after optimizations).

Removing all extra calls and therefore widening the compilation span is the biggest benefit of *GNFI*. If Graal optimizes the left graph from Figure 4, it can remove the `Object` array for the arguments. It can also remove the boxing of the primitive value 1.5. The right graph of Figure 4 shows the final graph after all optimizations. All argument manipulations could be eliminated, only the native call itself remains.

One might argue that it is unlikely that an installed code is a compile-time constant. However, for call stubs of native target functions, their installed code is nearly always a compile-time constant.

The call stub only depends on the signature of the native target function. If the signature is known at compile-time, which is the case for most native target functions, the call stub graph can be indeed seen as a compile-time constant. The only case where a call stub is not a compile-time constant, is when the signature contains a variable number of parameters. In this case *GNFI* has to create specific call stubs for different calls at run-time.

The main advantage of our approach compared to other approaches such as JNI or JNA is that *GNFI* can remove all extra call sites enabling the JIT compiler to optimize the argument conversion code.

6 Evaluation

We evaluate *GNFI* against JNI and JNA. The first benchmark is a micro benchmark that determines the cost of a call. It invokes an empty C function (with no arguments) in a loop. We chose this benchmark to measure the call overhead of the three approaches.

Secondly, we benchmark an application that uses a native numerical library, accessed by JNI, JNA and *GNFI*. We decided to use this benchmark, since researchers in the past extensively investigated aspects such as efficiency and usability when calling native numerical libraries [12, 13, 14, 15]. None of these investigations considered the extra calling overhead caused by JNI and JNA.

As a concrete application that uses a native numerical library we chose jblas [7], a Java wrapper for BLAS, which is the de-facto industry standard for matrix operations. For the evaluation we replaced all usages of JNI by *GNFI* or JNA. The evaluation compares these three approaches.

The charts in this section are all arranged on a linear, unbiased, higher-is-better scale. The y-axis shows the different approaches that we evaluated. The x-axis shows the number of executed calculations performed within 10 seconds. For all of the benchmark results in this section, we executed the benchmark 10 times with the same parameters. We calculate the averages for specific benchmarks using the arithmetic mean, and the associated charts also show the minimum and the maximum of the 10 runs for each benchmark.

The benchmarks were executed on an Intel Core i7 3520M dual-core 2.9 GHz CPU running 64 Bit Fedora 17 (Linux3.6.9-2) with 8 GB of memory. *GNFI* is built on Graal revision `ee8cd087a731` from

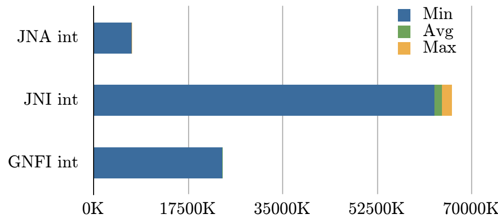


Figure 5: Performance of *nopFunction* (interpreted mode)

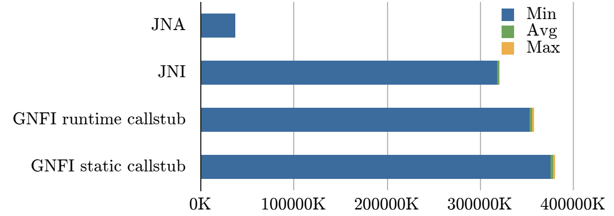


Figure 6: Performance of *nopFunction* (compiled mode)

the official OpenJDK Graal repository[1]. We used OpenJDK 1.7.0_09 for JNI, and JNA version 3.5.2.

The evaluation of each benchmark contains results for the following configurations: In interpreted mode *int* we executed the benchmarks with compilation disabled, thus with all code running in the interpreter. This mode uses *GNFI* as we described in Section 5 and compilation disabled. For JNI and JNA we also disabled the compilation in the VM.

In the compiled mode we measure the performance after an initial warm-up with compilation enabled, using the Graal VM. For *GNFI* we distinguish two modes of operation. In the first mode (*runtime callstub*) we make sure that our optimizations cannot inline the call stub. Therefore, *GNFI* has to call it. In the second mode (*static callstub*) the *GNFI* version inlines the call stub. Section 5 describes the configuration for *GNFI* in detail. For the JNI and the JNA versions of the benchmarks we also measure the performance after an initial warm-up with compilation enabled (using the Graal VM).

In the following sections we briefly describe each benchmark and evaluate it in the three modes.

6.1 Microbenchmarks

To measure the pure call overhead, we create a shared library that contains an empty function (*nopFunction*) without any parameters. We count the number of invocations possible within 10 seconds.

Figure 5 shows the comparison of *GNFI*, JNA and JNI when calling *nopFunction* in interpreted mode. We can see that *GNFI* is slower than JNI by a factor of 2.7. However, it is still faster than JNA by a factor of 3.4.

Compared to JNI, *GNFI* has to go through more layers to perform the call. As we described in Section 5, *GNFI* has to bridge interpreted and compiled code, which impedes the performance. For the performance of an application, interpreted code plays a minor role. Hence, when we designed our implementation, we neglected the performance in interpreted mode and focused on performance in compiled mode.

Figure 6 shows the comparison of JNI, JNA and *GNFI* on *nopFunction* in compiled mode. *GNFI* outperforms JNA by a factor of 10.3. It is now also faster than JNI by a factor of 1.2 with the *static callstub*. The performance of the *static callstub* is 6% better than the performance of the *runtime callstub*.

For the micro benchmarks, *GNFI* with *static callstub* can inline the callstub (described in Section 5) and thus remove one call site. This inlining explains the 6% speedup of the *static callstub* compared to the *runtime callstub*. *GNFI* with *runtime callstub* is faster than JNI because *GNFI* does a direct call to the call stub. Compared to JNI, *GNFI* does not set up any environment parameters, which explains the different performances.

6.2 Jblas Matrix Multiplication Benchmark

The jblas benchmark measures how often random double matrices of different sizes can be multiplied within 10 seconds. The matrix dimensions are $n \times n$ where $n \in \{10, 100, 1000\}$.

In interpreted mode *int* for $n = 10$ (Figure 7, upper three bars), *GNFI* is slower by a factor of 2.6 compared to JNI. Compared to the JNA, the *GNFI* is 12X faster.

We explain the slowdown of *GNFI* compared to JNI in interpreted mode by the fact that *GNFI* has to go through more layers to perform the call as explained in the evaluation of the micro benchmarks. However, when increasing the problem size, *GNFI* performs better than JNI even in interpreted mode. The benchmark results in Figure 8 show the comparison for $n = 100$, where *GNFI* outperforms JNI by a factor of 1.7. *GNFI* is 2X faster than JNA for $n = 100$. We explain the smaller performance gap between JNA and *GNFI* by a constant call overhead of JNA which has less impact when more time is spent inside the native method.

[1]available at `http://hg.openjdk.java.net/graal/graal`

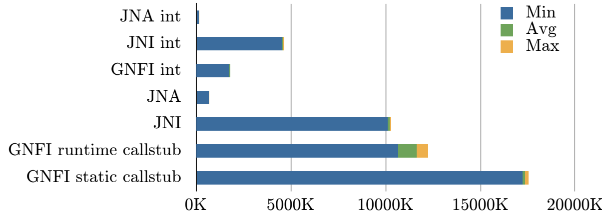


Figure 7: Performance of *jblas* for $n = 10$

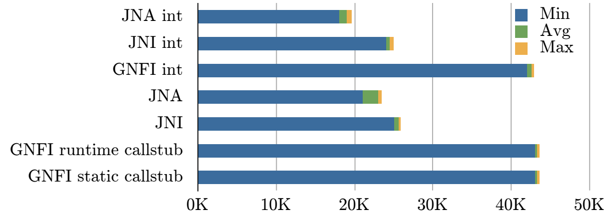


Figure 8: Performance of *jblas* for $n = 100$

Similar for $n = 1000$ (Figure 9) *GNFI* beats JNI by a factor of 1.9. *GNFI* is also faster than JNA by a factor of 1.9. The JNI jblas wrapper and the JNA interface copy the elements of the matrices to perform the call to the BLAS native target function.

Figure 9: Performance of *jblas* for $n = 1000$

In our implementation we directly pass a reference to the matrices on the Java heap. We can do this by passing the Java array object to the call stub. The call stub resolves the address of the array data and passes this pointer to the native target function. *GNFI* benefits from the fact that jblas represents matrices as one-dimensional Java arrays.

In compiled mode *GNFI* is faster than JNI and JNA for all problem sizes. In mode *runtime callstub* we intrinsify the function handle of *GNFI* as described in Section 5. The caller method makes a direct call to the call stub. Thus, *GNFI* is faster than JNI by a factor of 1.14 for $n = 10$ (Figure 7), by a factor of 1.7 for $n = 100$ (Figure 8), and by a factor of 1.9 for $n = 1000$ (Figure 9). *GNFI* is faster than JNA by a factor of 17 for $n = 10$, by a factor of 1.8 for $n = 100$, and by a factor of 1.9 for $n = 1000$. Again, the performance gap between JNA and *GNFI* gets smaller with greater n because constant call overhead of JNA has less impact.

In mode *static callstub*, the machine code of the benchmark can even call the BLAS function directly. However, the optimization from *runtime callstub* to *static callstub* only has an effect on $n = 10$ where the call overhead still counts. For $n = 100$ and $n = 1000$ the speedup is not noticeable, because the constant call overhead, compared to the time spent in the BLAS function, becomes negligible. The performance advantage of *GNFI* results from not having to copy the matrices.

While for $n = 10$ there was still a performance gap between compiled and interpreted Java code, this performance gap becomes almost non-existent for $n = 100$ and $n = 1000$.

7 Related Work

From the user's perspective, JNA is similar to our implementation. It allows the programmer to dynamically invoke native target functions without having to write native code. The programmer can simply dispatch the call from Java. As with *GNFI*, one loads a library and can then call arbitrary native target functions contained in it. JNA offers the same flexibility as *GNFI* in terms of dynamically calling native target functions. It uses a native library stub to dynamically invoke the native target code. However, as with JNI, the JIT compiler cannot inline this native code. Therefore, JNA cannot compete with *GNFI* in terms of performance as our evaluation in Section 6 shows.

Hirzel and Grimm [16] propose another approach which tries to simplify the invocation of C code from Java, called Jeannie, which integrates Java and C code. It is both a language and a compiler to allow both Java and C code in the same file. The programmer can easily combine the two languages without the boiler-plate code of JNI. However, since it is translated to JNI code it suffers from the same performance problems as the JNI.

Stepanian et al. [17] try to widen the compilation span by directly inlining native code by the JIT compiler. Their approach converts the native code to the compiler's intermediate language on which the JIT compiler then performs optimizations. Following their approach of decompiling and inlining native JNI wrapper functions, they could gain a significant speedup compared to JIT-compiled code without inlining. However, their approach differs from our approach because they do not provide an alternative to JNI. The approach of Stepanian et al. speeds up the JNI while preserving the portability properties of the JNI. The programmer still uses JNI to access native target functions, though their JIT compiler is able to inline the native code of JNI. With *GNFI* we provide a way to invoke native target functions

from Java directly while improving performance and also usability.

The approach used in the Compiled Native Interface (CNI) [18] is based on the idea of making GNU Java compatible with GNU C++. CNI allows writing Java native methods in C++ with zero overhead. Hooks in G++ (GNU C++ compiler) are used so that C++ code can access Java objects as naturally as native C++ objects. The approach of CNI is faster then JNI, but a less portable. It differs from our approach because *GNFI* provides facilities to call native target functions from Java, thus *GNFI* avoids writing native code at all.

8 Conclusion

Current methods of calling native code from Java are not as straightforward as they could be. Using JNI, for example, the programmer has to write wrappers in native code, compile and link libraries to them. Furthermore, interfaces like JNI suffer from performance problems.

In this paper we proposed a new native function interface, *GNFI*. Compared to similar approaches, *GNFI* has advantages in terms of usability and performance. In terms of usability, the programmer does not have to write C/C++ glue code to call native target functions. Thus, the programmer does not have to compile such source files or to link them against libraries. As we showed in this paper, it is possible and beneficial to implement a C calling convention on the Java side.

We have demonstrated the performance of *GNFI* for the jblas library which is a Java wrapper for the native vector and matrix operation library BLAS. We used jblas to evaluate a matrix multiplication benchmark. The results show that for calling native code from Java *GNFI* performs better than JNI and JNA in all relevant cases.

For native functions with a fixed number of parameters the JIT-compiled code of the application contains direct calls to these native target functions. For functions with a variable number of parameters we first invoke the call stub which invokes the native target function.

Programmers can use *GNFI* wherever they would have written a JNI wrapper for calling native target functions before. Another class of applications which can benefit from *GNFI* are language interpreters on top of the Java VM. One example of this idea are the Truffle language implementations [1, 2] where a C or Python interpreter can call C libraries or system functions using *GNFI*.

References

[1] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer, "Self-optimizing ast interpreters," in *Proceedings of the 8th symposium on Dynamic languages*, pp. 73–82, ACM, 2012.

[2] C. Wimmer and T. Würthinger, "Truffle: a self-optimizing runtime system," in *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pp. 13–14, ACM, 2012.

[3] R. Gordon, *Essential JNI: Java Native Interface*. Prentice-Hall, Inc., 1998.

[4] S. Liang, *The Java Native Interface: Programmer's Guide and Specification*. 1999.

[5] D. Kurzyniec and V. Sunderam, "Efficient cooperation between java and native codes–jni performance benchmark," in *The 2001 International Conference on Parallel and Distributed Processing Techniques and Applications*, Citeseer, 2001.

[6] Oracle, "OpenJDK: Graal project." `http://openjdk.java.net/projects/graal/`, 2013.

[7] L. Braun, Mikio, "jblas - linear algebra for java." `http://jblas.org/`, 2013.

[8] G. Tan and J. Croft, "An empirical security study of the native code in the jdk," in *Usenix Security Symposium (SS)*, 2008.

[9] L. Stadler, G. Duboscq, H. Mössenböck, and T. Würthinger, "Compilation queuing and graph caching for dynamic compilers," in *Proceedings of the sixth ACM workshop on Virtual machines and intermediate languages*, VMIL '12, (New York, NY, USA), pp. 49–58, ACM, 2012.

[10] G. Duboscq, L. Stadler, T. Würthinger, D. Simon, and C. Wimmer, "Graal IR: An extensible declarative intermediate representation," in *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*, 2013.

[11] Oracle, "Graal openjdk project documentation." `http://lafo.ssw.uni-linz.ac.at/javadoc/graalvm/all/index.html`, 2013.

[12] V. Getov, S. Hummel, Flynn, and S. Mintchev, "High-performance parallel programming in java: Exploiting native libraries," *Concurrency: Practice and Experience*, vol. 10, no. 11-13, pp. 863–872, 1998.

[13] B. Blount and S. Chatterjee, "An evaluation of java for numerical computing," in *Computing in Object-Oriented Parallel Environments*, pp. 35–46, Springer, 1998.

[14] A. J. C. Bik and D. B. Gannon, "A note on native level 1 blas in java," *Concurrency: Practice and Experience*, vol. 9, no. 11, pp. 1091–1099, 1997.

[15] M. Baitsch, N. Li, and D. Hartmann, "A toolkit for efficient numerical applications in java," *"Advances in Engineering Software"*, vol. 41, no. 1, pp. 75 – 83, 2010.

[16] M. Hirzel and R. Grimm, "Jeannie: granting java native interface developers their wishes," in *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, (New York, NY, USA), pp. 19–38, ACM, 2007.

[17] L. Stepanian, A. Brown, Demke, A. Kielstra, G. Koblents, and K. Stoodley, "Inlining java native calls at runtime," in *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, VEE '05, (New York, NY, USA), pp. 121–131, ACM, 2005.

[18] "Cni (compiled native interface)." `http://gcc.gnu.org/onlinedocs/gcj/About-CNI.html`, 2013.

Control Flow Unfolding of Workflow Graphs Using Predicate Analysis and SMT Solving

Thomas S. Heinze[1], Wolfram Amme[1], and Simon Moser[2]

[1] Friedrich Schiller University of Jena
[t.heinze,wolfram.amme]@uni-jena.de
[2] IBM Research & Development Boeblingen
smoser@de.ibm.com

Abstract. We present an extension of our previously introduced technique for unfolding conditional control flow in extended workflow graphs. This technique allows for a more precise process-to-Petri-net-mapping which is crucial for business process verification. Our new technique derives data flow information about the state space of process data by means of predicate clauses using a novel CSSA-Form-based analysis. The derived information is then exploited in an adjusted unfolding algorithm to resolve conditional control flow utilising the SMT solver YICES.

1 Introduction

Verification of business processes today is typically done using Petri-net-based process models, which allows for a natural modeling and analysis of vital aspects like parallelism and message exchange. The quality of verification thereby strongly depends on the precision of the process-to-Petri-net mapping. In particular, there exists an inherent tradeoff between verification effectivity and precision, as typical properties for business process verification, e.g., soundness or controllability, are in general undecidable for full-specified business processes.

For this reason, more often than not, methods for business process verification omit process data so that the used Petri-net-based process models merely represent the processes' (unconditional) control flow. By this means, the conditional control flow of a process, i.e., its data-dependent branchings and loops, is mapped to nondeterminism which only over-approximates the process's actual behaviour (under the fairness assumption). However, as research has shown lately, such an approach comprises the danger of an erroneous verification, by means of both, false-positive and false-negative verification results [2,7].

In order to tackle this problem, in our previous work [2,3], we have developed a *control flow unfolding technique* which allows for an increase in the precision of the process-to-Petri-net mapping. More specifically, the developed technique transforms certain kinds of conditional control flow into unconditional control flow for a business process, without inferring the process's execution semantics. As a result, when mapping a thus preprocessed process to its Petri net model, there is no need to over-approximate conditional control flow using nondeterminism and therefore no possible source of error for verification. In other words, we have

envisioned a compiler, which takes as input a business process and generates as output a Petri net. However, in contrast to a conventional compiler, its objective is not to result in efficient runtime code but rather to produce a most-precise though still effectively verifiable Petri-net-based process model.

Our previous technique exploited data flow information derived by copy propagation [2], i.e., information about constant values, or value range analysis [3], i.e., value ranges for integers, to unfold a process's conditional control flow. In this work, we will sketch a new *CSSA-Form-based analysis* for gaining a more general representation for the state space of process data in terms of predicate clauses and how the such derived information is then used to integrate a *SMT solver* into our unfolding approach, so that its applicability is further widened.

The remainder of the paper is structured as follows: The following section introduces the example process which is used for illustration throughout the paper. In Section 3, we describe the CSSA-Form-based analysis to derive predicate clauses. The use of the thus derived data flow information in our adjusted control flow unfolding technique is explained in Section 4. Finally, after a brief discussion of related work in Section 5, Section 6 concludes the paper.

2 Running Example: Rock-paper-scissors

For illustration, we will use the example shown in Figure 1. On its upper left-hand side, a (business) process is shown in a textual format. The process models the game *Rock-paper-scissors*, where two partners (A and B) play against each other. The idea is, that each player decides whether to take one of the three items: rock, paper, scissors. If A takes scissors and B paper, A takes paper and B rock, or A takes rock and B scissors, player A wins the game and vice versa. If both players take the same item, the game continues with another round.

In order to implement the game, the three items are encoded in the process by using three integers, i.e., scissors becomes 0, paper becomes 1, rock becomes 2. In consequence, the decision whether player A, who has chosen $\$a$, won over player B, who has chosen $\$b$, can be done based on the expression $(\$a + 1)\ mod\ 3 = \$b$ and vice versa. Therefore, the process contains a loop which tests if A and B chose the same item. If so, the loop continues and another round is played. In the loop, A and B state their choice by sending an integer to the process, which is encoded into either 0, 1, or 2. Afterwards, the winner is determined, if existent, using two conditional branchings and an appropriate message is sent back.

When verifying this process with regard to *controllability* [4], i.e., whether partners A and B exist for which the process will always be able to terminate its execution, the process is mapped to a Petri-net-based process model first. The application of a conventional mapping, e.g., using the pattern-based approach described in [4,5], results in the Petri net shown in Figure 1. Note that the loop and conditional branchings are therein mapped to conflicting transitions modeling nondeterminism. Thus, the Petri net only over-approximates the process's control flow such that verifying the process based on this process model results in an erroneous finding that the process is not controllable, while it rather is.

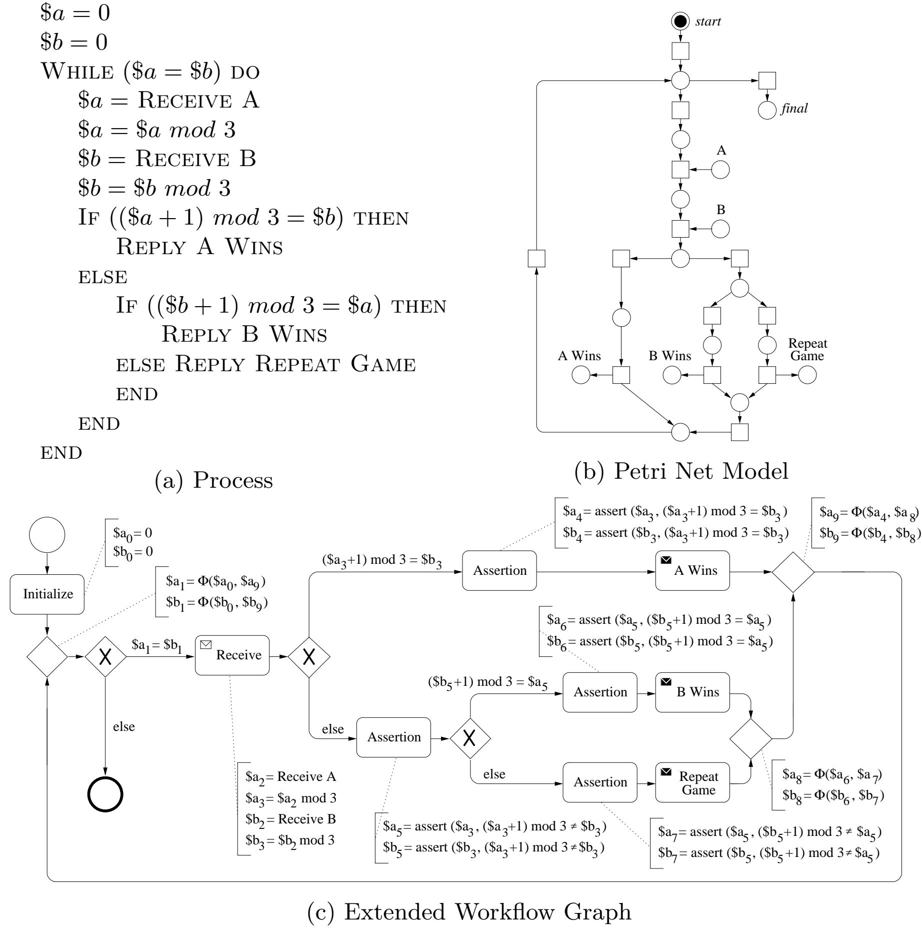


Fig. 1. Running example: Process, Petri net model, and extended workflow graph

3 Predicate Clause Analysis

Using control flow unfolding as described in [2,3], allows for resolving certain kinds of conditional control flow such that nondeterminism can be avoided in a process's Petri net model. To this end, data flow information is derived and used to identify control flow paths where a loop or branching condition is statically evaluable based on the gained information. These control flow paths are then made explicit, i.e., unfolded, and, as a result, the condition can be removed.

In order to derive data flow information about process data, we use *extended workflow graphs* [2]. Workflow graphs provide a well-known format to model the control flow structure of a process. For the representation of process data, workflow graphs are enriched by annotating nodes and edges with instructions and condition expressions in *Concurrent Static Single Assignment (CSSA-) Form* [1], which benefits analysis. The such defined extended workflow graph for

our example process is shown at the bottom of Figure 1. Note that the process's variables are therein (statically) defined only once such that variables become values, which is denoted by subscripts, e.g., $\$a$ becomes $\$a_0, \ldots, \$a_9$. In order to merge conflicting variable definitions into a single value, *Φ-functions* are used, as is done for variable $\$a$'s and $\$b$'s definitions, e.g., $\$a_1 = \Phi(\$a_0, \$a_9)$. Further, several *assertions* have been added exposing the induced constraints for a value referenced in a condition expression, e.g., $\$a_4 = assert(\$a_3, (\$a_3+1)\ mod\ 3 = \$b_3)$ guarantees for all dominated uses of value $\$a_3$, which have been renamed to $\$a_4$, that its value satisifies the branching condition $(\$a_3 + 1)\ mod\ 3 = \$b_3$.

For the example process, data flow information resulting from constant propagation or value range analysis does not help control flow unfolding since the information derivable is too limited to allow for evaluating the loop or branching conditions. In contrast, we here employ a novel analysis based on the analysis framework described in [1], which produces a more general notion for the state space of process data in terms of predicates. Thereby, predicates denote instructions or condition expressions as they appear in the process's extended workflow graph. Sets of predicates depict conjunctions of predicates, so-called *predicate clauses*, as they hold on a single control flow path. Sets of predicate clauses are then used, by means of disjunctions, to merge information over multiple paths.

In the following, let *Variables* denote the set of variables and *Predicates* the set of instructions and condition expressions appearing in an extended workflow graph. *Predicates* is augmented with instructions in $\{x = y \mid x, y \in Variables\}$. Function $var(pred)$ returns the set of contained variables for $pred \in Predicates$. Then, for each variable v, we estimate its state space in terms of sets of predicate clauses $inf(v)$ using the following CSSA-based data flow equations:

Incoming Message If variable v is defined by incoming message activity, e.g., Receive, the result is the singleton set $inf(v) = \{\emptyset\}$

Constant Assignment If variable v is defined by constant assignment, i.e., $v = c$, the result is the singleton set $inf(v) = \{\{v = c\}\}$

General Assignment If variable v is defined by expression assignment $v = expr$ with $var(expr) = \{x_1, \ldots, x_n\}$, the result is:

$$inf(v) = \bigcup_{k_1 \in inf(x_1), \ldots, k_n \in inf(x_n)} \{k_1 \setminus kill(v) \cup \ldots \cup k_n \setminus kill(v) \cup \{v = expr\}\}$$

Assertion If variable v is defined by assertion, i.e., $v = assert(x, pred)$ with $var(pred) = \{x_1, \ldots, x_n\}$, the result is:

$$inf(v) = \bigcup_{k_1 \in inf(x_1), \ldots, k_n \in inf(x_n)} \{k_1 \setminus kill(v) \cup \ldots \cup k_n \setminus kill(v) \cup \{pred, v = x\}\}$$

Φ-/π-Function If variable v is defined by Φ-/π-function, i.e., $v = \Phi(x_1, \ldots, x_n)$ or $v = \pi(x_1, \ldots, x_n)$, the result is:

$$inf(v) = \bigcup_{k_i \in inf(x_i)} \{k_i \setminus kill(v) \cup \{v = x_i\}\}$$

where $kill(v) = \{pred \in Predicates \mid v \in var(pred)\}$ for all $v \in Variables$.

Applying the analysis to the example process then results for variable $\$a_1$:

$$\{ \{ \$a_0 = 0 \ , \ \$a_1 = \$a_0 \ \},$$
$$\{ \$a_3 = \$a_2 \ mod \ 3 \ , \ \$b_3 = \$b_2 \ mod \ 3 \ , \ (\$a_3 + 1) \ mod \ 3 = \$b_3 \ , \ \$a_4 = \$a_3,$$
$$\$a_9 = \$a_4 \ , \ \$a_1 = \$a_9 \ \},$$
$$\{ \$a_3 = \$a_2 \ mod \ 3 \ , \ \$b_3 = \$b_2 \ mod \ 3 \ , \ (\$a_3 + 1) \ mod \ 3 \neq \$b_3 \ , \ \$a_5 = \$a_3,$$
$$\$b_5 = \$b_3, (\$b_5 + 1) \ mod \ 3 = \$a_5, \$a_6 = \$a_5, \$a_8 = \$a_6, \$a_9 = \$a_8, \$a_1 = \$a_9 \ \},$$
$$\{ \$a_3 = \$a_2 \ mod \ 3 \ , \ \$b_3 = \$b_2 \ mod \ 3 \ , \ (\$a_3 + 1) \ mod \ 3 \neq \$b_3 \ , \ \$a_5 = \$a_3,$$
$$\$b_5 = \$b_3, (\$b_5 + 1) \ mod \ 3 \neq \$a_5, \$a_7 = \$a_5, \$a_8 = \$a_7, \$a_9 = \$a_8, \$a_1 = \$a_9 \ \} \ \}$$

and for variable $\$b_1$:

$$\{ \{ \$b_0 = 0 \ , \ \$b_1 = \$b_0 \ \},$$
$$\{ \$a_3 = \$a_2 \ mod \ 3 \ , \ \$b_3 = \$b_2 \ mod \ 3 \ , \ (\$a_3 + 1) \ mod \ 3 = \$b_3 \ , \ \$b_4 = \$b_3,$$
$$\$b_9 = \$b_4 \ , \ \$b_1 = \$b_9 \ \},$$
$$\{ \$a_3 = \$a_2 \ mod \ 3 \ , \ \$b_3 = \$b_2 \ mod \ 3 \ , \ (\$a_3 + 1) \ mod \ 3 \neq \$b_3 \ , \ \$a_5 = \$a_3,$$
$$\$b_5 = \$b_3, (\$b_5 + 1) \ mod \ 3 = \$a_5, \$b_6 = \$b_5, \$b_8 = \$b_6, \$b_9 = \$b_8, \$b_1 = \$b_9 \ \},$$
$$\{ \$a_3 = \$a_2 \ mod \ 3 \ , \ \$b_3 = \$b_2 \ mod \ 3 \ , \ (\$a_3 + 1) \ mod \ 3 \neq \$b_3 \ , \ \$a_5 = \$a_3,$$
$$\$b_5 = \$b_3, (\$b_5 + 1) \ mod \ 3 \neq \$a_5, \$b_7 = \$b_5, \$b_8 = \$b_7, \$b_9 = \$b_8, \$b_1 = \$b_9 \ \} \ \}$$

Note that the *Φ*-functions and assertions are included by means of simple assignments, each copying the respective operand's value to the function value.

4 Control Flow Unfolding

Having done the analysis, the derived data flow information, i.e., predicate clauses, can be tested for enabling the evaluation of conditional control flow. Thus, given a branching or loop condition, a *SMT solver*, in our case *YICES*[3], is used to check, on the one hand, whether a conjunction of certain predicate clauses for variables referenced in the condition is satisfiable (otherwise it would represent an infeasible path and can be neglected) and, on the other hand, whether the conjunction implies the condition to be either true or false. In the latter, the condition can be statically evaluated for this specific set of predicate clauses and is thus a candidate for control flow unfolding.

In the example process, the loop with condition $\$a_1 = \$b_1$ is such a candidate for unfolding, i.e., the control flow path which is denoted by the predicate clause $\{\$a_3 = \$a_2 \ mod \ 3, \$b_3 = \$b_2 \ mod \ 3, (\$a_3 + 1) \ mod \ 3 \neq \$b_3, \$a_5 = \$a_3, \$b_5 = \$b_3, (\$b_5 + 1) \ mod \ 3 \neq \$a_5, \$a_7 = \$a_5, \$a_8 = \$a_7, \$a_9 = \$a_8, \$a_1 = \$a_9\}$ for variable $\$a_1$ and the clause $\{\$a_3 = \$a_2 \ mod \ 3, \$b_3 = \$b_2 \ mod \ 3, (\$a_3 + 1) \ mod \ 3 \neq \$b_3, \$a_5 = \$a_3, \$b_5 = \$b_3, (\$b_5 + 1) \ mod \ 3 \neq \$a_5, \$b_7 = \$b_5, \$b_8 = \$b_7, \$b_9 = \$b_8, \$b_1 = \$b_9\}$ for variable $\$b_1$ is a feasible path since the conjunction of both clauses is satisfiable. Further, the conjunction of the clauses also implies the loop condition $\$a_1 = \$b_1$ always to be satisfied, as can be checked using YICES:

$$\models \ (\$a_3 = \$a_2 \ mod \ 3 \wedge \$b_3 = \$b_2 \ mod \ 3 \wedge (\$a_3 + 1) \ mod \ 3 \neq \$b_3 \wedge \$a_5 = \$a_3$$
$$\wedge \$b_5 = \$b_3 \wedge (\$b_5 + 1) \ mod \ 3 \neq \$a_5 \wedge \$a_7 = \$a_5 \wedge \$b_7 = \$b_5 \wedge \$a_8 = \$a_7$$
$$\wedge \$b_8 = \$b_7 \wedge \$a_9 = \$a_8 \wedge \$b_9 = \$b_8 \wedge \$a_1 = \$a_9 \wedge \$b_1 = \$b_9) \rightarrow \$a_1 = \$b_1$$

[3] `http://yices.csl.sri.com`

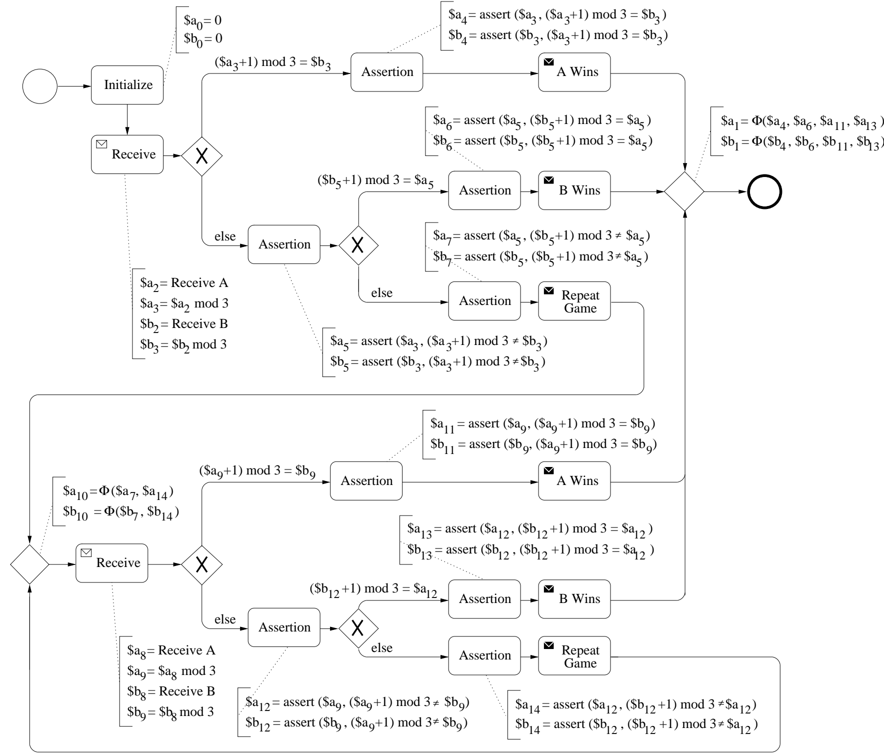


Fig. 2. Extended workflow graph with unfolded loop

Since the derived predicate clauses make it possible to evaluate the loop condition to either true or false for all control flow paths in the example process, the loop can be effectively transformed such that the loop condition is removed. To this end, the control flow paths which are associated to the individual predicate clauses are made explicit by dissolving merge nodes joining these paths through subgraph duplication. In particular, the loop is replaced by copies of it, so-called *loop instances*, based on the derived predicate clauses which therefore act as a kind of invariant for the values of variables $\$a_1$ and $\$b_1$. For instance, predicate clauses $\{\$a_0 = 0, \$a_1 = \$a_0\}$ and $\{\$b_0 = 0, \$b_1 = \$b_0\}$ constitute the invariant $\$a_0 = 0 \wedge \$a_1 = \$a_0 \wedge \$b_0 = 0 \wedge \$b_1 = \$b_0$ which holds for the first loop iteration and allows for evaluating the loop condition to true therein. Thus, the loop condition can be evaluated in each loop instance and afterwards replaced by unconditional control flow to the loop exit or to the same or another instance.

For conducting the above described *control flow unfolding technique*, an adjusted version of the algorithm described in [3] is used, which works with predicate clauses as data flow information and evaluates loop and branching conditions by the help of SMT solver YICES. Applying this algorithm to the loop in the example process results in the extended workflow graph shown in Figure 2. As can be seen, the loop is therein unfolded into two loop instances such that

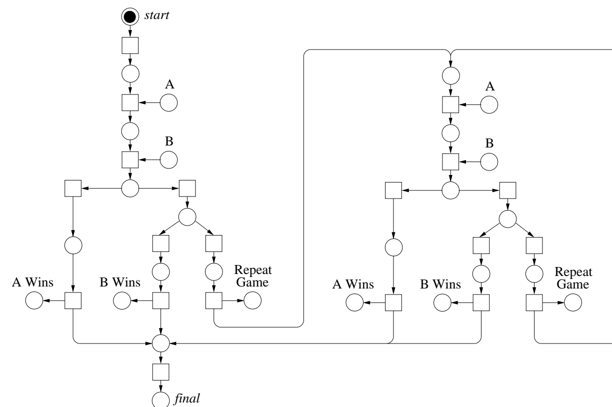


Fig. 3. Refined Petri net model

the loop condition has been eventually resolved. Mapping the thus successfully unfolded process to a Petri-net-based process model yields the Petri net shown in Figure 3. Verifying the process in respect of controllability based on this refined process model then comes to the correct result that the process is controllable.

5 Related Work

The relevance of process data when verifying business processes based on Petri nets is an ongoing research topic. Nevertheless, most approaches to a process-to-Petri-net-mapping either omit data entirely our restrict themselves to data of bounded and limited domain [4,5,7]. Using high-level Petri nets allows for augmenting the process model with (unbounded) data, for which verification methods have been proposed in respect of acyclic processes. However, the application of high-level nets in general leads to undecidability in case of cyclic control flow, and even if data is bounded, state space explosion may hinder a feasible verification. This also applies if high-level nets are unfolded into low-level Petri nets, since an infinite data domain implies an infinite low-level net. In contrast, our unfolding technique always terminates with a finite process model.

A method to integrate SMT solving into Petri-net-based business process verification has already been described in [6]. However, this approach is also restricted to acyclic processes since its termination can else not be guaranteed.

6 Conclusion

In this paper, we presented an extension of our previous technique [3], which allows us to unfold certain kinds of a business process's conditional into unconditional control flow such that a precise mapping of the process to its Petri net model is not impeded by the introduction of nondeterminism. In our previous work, we

have based the control flow unfolding on data flow information derived by copy propagation or value range analysis. However, the information thus derivable can be too limited for resolving certain loop or branching conditions, as is shown by the running example in this paper. In order to enlargen the number of cases our technique is effectively applicable, we now employ a CSSA-based analysis for deriving predicates determining the state space of process data, which are then used in combination with a SMT solver to conduct the unfolding. In this way, we are able to exploit the various background theories available in SMT solvers (supporting real/integer arithmetic, arrays, lists, etc.).

In a current prototype, we have implemented the unfolding technique for a subset of the WS-BPEL language based on value range information. Using the prototype in combination with existing Petri-net-based verification tools, e.g., Fiona [4], then allows for achieving more precise verification results. We plan to integrate the predicate clause analysis and adjusted unfolding algorithm described here into this prototype. Building on that, the thorough evaluation of the control flow unfolding approach remains the main issue for future work.

References

1. 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), 47–59 (2009)
2. Heinze, T.S., Amme, W., Moser, S.: A Restructuring Method for WS-BPEL Business Processes Based on Extended Workflow Graphs. In: Dayal, U., Eder, J., Koehler, J., Reijers, H.A. (eds.) Business Process Management, 7th International Conference, BPM 2009, Ulm, Germany, September 8-10, 2009, Proceedings. pp. 211–228. No. 5701 in Lecture Notes in Computer Science, Springer (2009)
3. Heinze, T.S., Amme, W., Moser, S., Gebhardt, K.: Guided Control Flow Unfolding for Workflow Graphs Using Value Range Information. In: Schönberger, A., Kopp, O., Lohmann, N. (eds.) Services and their Composition, 4th Central European Workshop on Services and their Composition, 4. Zentral-europäischer Workshop über Services und ihre Komposition, ZEUS 2012, Bamberg, February 23.-24. 2012, Post-Workshop Proceedings. pp. 128–135. No. 847 in CEUR Workshop Proceedings, CEUR-WS.org (2012)
4. Lohmann, N., Massuthe, P., Stahl, C., Weinberg, D.: Analyzing interacting WS-BPEL processes using flexible model generation. Data & Knowledge Engineering 64(1), 38–54 (2008)
5. Lohmann, N., Verbeek, E., Ouyang, C., Stahl, C.: Comparing and evaluating Petri net semantics for BPEL. International Journal of Business Process Integration and Management 4(1), 60–73 (2009)
6. Monakova, G., Kopp, O., Leymann, F.: Improving Control Flow Verification in a Business Process using an Extended Petri Net. In: Kopp, O., Lohmann, N. (eds.) Services und ihre Komposition, Erster zentraleuropäischer Workshop, ZEUS 2009, Stuttgart, 2.-3. März 2009, Proceedings. pp. 95–101. No. 438 in CEUR Workshop Proceedings, CEUR-WS.org (2009)
7. Sidorova, N., Stahl, C., Trčka, N.: Soundness verification for conceptual workflow nets with data: Early detection of errors with the most precision possible. Information Systems 36(7), 1026–1043 (2011)

Specification of Real-Time Systems with Mixed Time-Criticality *

Raimund Kirner
University of Hertfordshire
United Kingdom
r.kirner@herts.ac.uk

Abstract

Models for real-time computing are available with different timing requirements. With the ongoing trend towards integration of services of different degrees of timing strictness on one single platform, there is a need to specify computing models for such scenarios.

In this paper we clarify the concept of mixed time-criticality *(MTC), which is a generalisation of real-time computing. This is based on the so-called* temporal service utility function*, which naturally combines the specification of timing properties and service criticalities. We derive the basic attributes that are needed to specify systems with MTC. These attributes can be used to develop MTC programming models and to study the scheduling of MTC systems. We present some examples to show the usefulness of MTC for real-life systems.*

1 Introduction

Computer systems with their timing behaviour being part of their correctness criterion are called real-time computer systems [8]. Soft real-time systems are considered to tolerate rare occurrences of deadline misses, while hard real-time systems may cause fatal consequences in case of deadline misses.

To cope with the increasing complexity of cyberphysical systems, application vendors are increasingly demanded towards solutions with mixed timing requirements and criticality levels of services. The research community has reacted by providing platforms with sufficient ways to realise temporal isolation of tasks in order to execute the jointly, regardless of having different timing requirements. An example is the *ACROSS* MPSoC (multiprocessor system-on-chip) platform which promotes time-triggered communication on an MPSoC network as the communication backbone [11]. The strong separation based on the time-triggered network supports isolation of cores, allowing also mixed-criticality integration. Another example is the *CompSOC* platform of Goossens et al., offering a virtual execution platform for each application running on it [4]. Design flows have been developed to map hard-real-time, soft-real-time, and non-real-time dataflow applications as well as Kahn process networks [7] onto the CompSOC platform. Composability for each resources is achieved by using preemptive time-division multiplexing (TDM) between applications for access to processor and NoC, which avoids interference between applications.

Chakraborty et al. have worked on mixed-criticality with timing and stability constraints based on FlexRay time-triggered networks [5]. They classify applications into two categories: (i) safety-critical control applications

*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). under contract no 295371 and the IST FP7 research project “Asynchronous and Dynamic Virtualization through performance ANalysis to support Concurrency Engineering (ADVANCE)”.

with stability and performance constraints, and (ii) time-critical applications with only deadline constraints. A schedule is constructed while at the same time optimising the sampling rate of the control applications.

Tamas-Selicean and Pop have worked on mixed criticality by considering hard-real-time tasks of different SIL [3] levels [13]. They need to solve (1) task-to-processor mapping, (2) task-to-partition assignment, (3) slot allocation at each processor, and (4) static schedule tables. Baruah et al. have also worked on different aspects of scheduling sporadic task sets with mixed criticality [2, 9, 1].

Summarising the state of the art, there has been done considerable amount of work on scheduling applications with different (timing) criticalities. There are also some MPSoC platforms proposed that support the engineering of such systems. What we found to be missing yet is a systematic concept of how to characterise system services of different time-criticality in a uniform way.

In this paper we present in Section 2 a uniform way of characterising hard-real-time, soft-real-time, and non-real-time system services, by using so-called *temporal service utility functions* (TSUF). Based on the TSUF characteristics, we propose different properties important for the definition of mixed time-criticality. Using these properties, we present a generic definition of mixed time-criticality. In Section 3 we describe what attributes a programming model should be able to provide in order to specify mixed time-criticality. In Section 4 we show some use cases for mixed time-criticality systems. Section 5 concludes this work.

2 Mixed Time-Criticality

In the following we clarify the concept of *mixed time-criticality*. A computer system typically has to provide different functionalities, which are usually summarised by its specification. We call these different functionalities *services*.

To clarify the meaning of a service, Figure 1 shows the relation between the terms *service*, *interface*, and *system*. The interface describes the system access by input and output. An interface might be specified at different levels of detail, for example, by an API, or by some abstract behaviour rules. A system is said to realise an interface. A service is a slice of the total system behaviour, with a subset of the system's interface used to access that service. Services stretch from the input to the output of a system, which is in contrast to a system component, which is a subsystem on its own, also realising its own interfaces and services. It is important to note that none of the concepts of Figure 1 are a synonym for the scheduling objects to be executed at runtime, as services may result in more than one scheduling object and services may also share some of their scheduling objects.

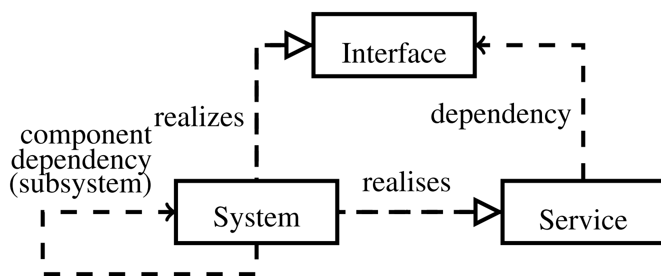


Figure 1. UML Relation between System and Services

2.1 Temporal Service Utility

Central for the concept of mixed time-criticality is the requirement of real-time services. The correctness of a real-time service requires the correct computation of the output values and also the correct timing of providing

the output. The latency denotes the timing interval between a trigger signal at the input and the corresponding response provided at the output. In classical real-time models a deadline is called *firm*, if the result is of no value after the deadline, otherwise the deadline would be called *soft*. If damaging failure can happen after a firm deadline has been missed, the deadline is called *hard* [8].

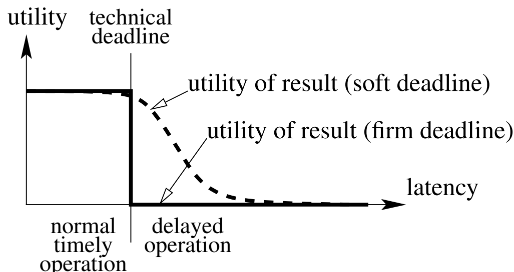


Figure 2. Utility of Result in Classical Real-Time Models

Figure 2 shows the temporal utility function of a result according to the classical real-time model. However, this model does not really reflect reality. Because in reality no one should chose a deadline such that after the deadline the utility of the result is immediately zero. Doing so would leave no safety margin, and furthermore, the relative distance from the chosen deadline and the precise point where the utility of the result could become zero is generally not known.

We propose a refined real-time model that acknowledges the difference of the latency where a result utility could become zero and the latency chosen as technical deadline. We describe the timing requirements of a real-time service by the *utility* of the computed result over the latency, which we call *temporal service utility function* (TSUF). Furthermore, the TSUF also reflects whether the given timing requirement is a hard deadline, by extending the utility also to negative values. Drawing the TSUF over lateny is just exemplary, one might equally well draw it for throughput or jitter. Drawing the TSUF over latency however is more self-evident, as each service response goes through a latency interval starting from zero till the time of response, while, for example, throughput cannot be understood by an individual service request.

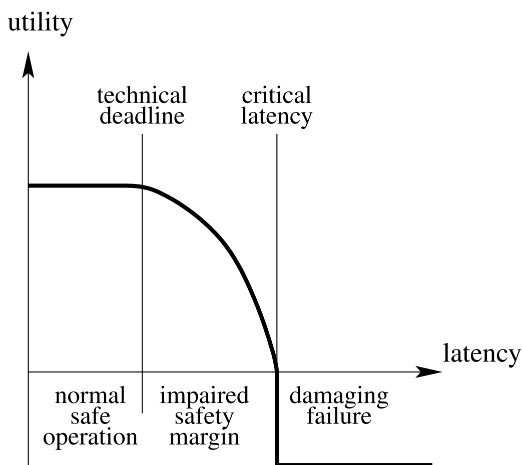


Figure 3. Temporal Service Utility Function of a Real-Time Service

A simple example of such a TSUF is shown in Figure 3. In this example the service utility is not only positive, but it also becomes negative after a certain time. A positive utility means that the system is able to provide a useful service. A negative utility means that the corresponding latency is outside the valid time range and as a consequence can result in a damage of the system. In the given example a service provided later than a certain time interval, denoted as *critical latency*, can provide system damage. This range of service latency is marked as *damaging failure* in Figure 3. The range of service latency marked as *normal safe operation* denotes the service latency considered as optimal operation, providing the maximum and constant service utility. There is also a latency range denoted as *impaired safety margin*, which does not yet cause any system damage, but exhibits a declining utility. The declining utility is a reflection of the reduced safety margin of the corresponding latency interval. When designing a system we usually set the deadline of a service to the end of the uncompromised service utility range, which we marked with *technical deadline* in the figure.

In the example the utility drops abruptly to full potential damage after the *critical latency*. However, in general a latency transition from *impaired safety margin* to *damaging failure* could also be more smooth, depending on the application. For example, with fuel injection in a motor there is a certain range of injection time where the motor becomes less efficient and is going to be increasingly worn out with later/earlier injection timing. In general, the transition from *normal safe operation* to *damaging failure* can be quite manifold, depending on the concrete real-time service. Research on real-time computing is most often oblivious of this diversity of transition characteristics of different real-time services.

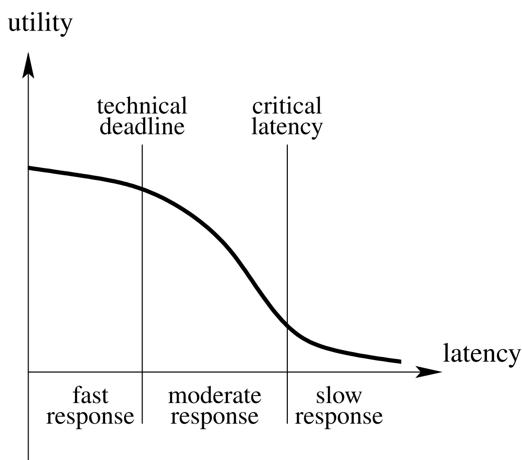


Figure 4. Temporal Service Utility Function of a Real-Time Service without Potential Damage

Characteristic for real-time services is that their TSUF shows a significant value reduction for some range of the latency values. Figure 3 shows a typical example of how the TSUF of a real-time service can look like. A negative TSUF value, as shown in the figure, is considered to represent damaging failure.

The existence of negative TSUF values is not a precondition for a service to be classified as a real-time service. For example, Figure 4 shows an example TSUF of a real-time service without any potential damage interval. However, the figure shows a significant variation among the TSUF values, which is a sufficient criterion to classify the service as real-time service.

On the contrary, to be classified as a non-real-time service, its TSUF may only exhibit a rather small variation. For example, the TSUF in Figure 5 can be considered as a non-real-time service, due to its small variation of the TSUF. There always will be a TSUF variation for any service, as no service is of use with an indefinitely long

response time. For exactly that reasons one may not be able to cast a strict separation between what is a real-time service and what is a non-real-time service. However, the comparison of the TSUF graph of Figure 4 and Figure 5 should make it sufficiently clear of when it is adequate to speak of a real-time service or of a non-real-time service.

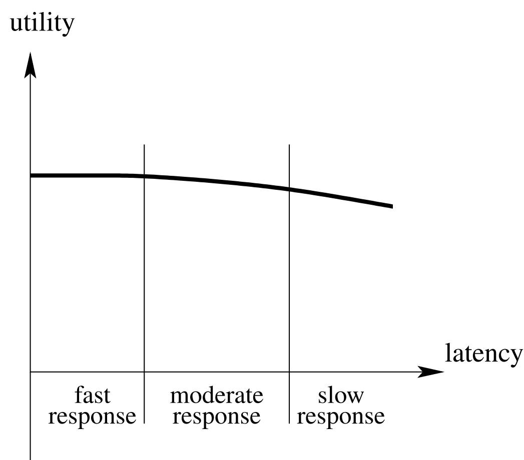


Figure 5. Temporal Service Utility Function of a Non-Real-Time Service

2.2 Performance Characteristics

System services are normally connected with requirements of their timing properties. Important timing properties of a service can be *throughput*, *latency*, and *jitter*.

The throughput describes how many input elements a system is able to process per time. The latency describes the delay between arrival of data at the system input and the release of the processed results at the system output. The jitter describes the variation of the latency.

These three performance metrics are the basic performance characteristics regardless of the concrete TSUF. However there is a difference on how the performance characteristics are being evaluated. For example, for real-time services it is important to consider the boundaries of the performance metrics. Foremost the minimum throughput, the maximum latency, and the maximum jitter are important to ensure the timeliness of a real-time service. It is worth noting that by binding the maximum latency and jitter, we implicitly also bind the minimum latency.

For non-real-time services it is typically the mean value of the performance characteristics that matters.

2.2.1 Primary Limits and Tolerance Ranges

Services can have besides their normal safe operation range another operation range of *impaired safety margin*, as shown by the TSUF in Figure 3. A subrange of this impaired safety margin might be used as a so-called tolerance range to extend the operability of the service for operation situations of unexpected resource shortage, etc. Figure 6 shows an example of a tolerance range of a real-time service's latency.

2.3 Deadline/Timing Strictness

Besides the specification of a service's performance metrics it is also useful to specify how strict the given performance metrics are meant to be. This information can help the system scheduler to decide for the best overall

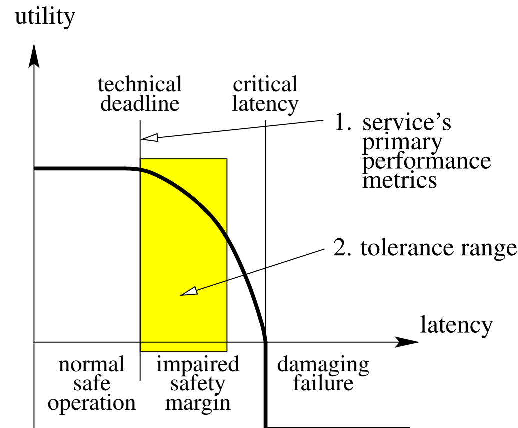


Figure 6. Example: Spcification of a Real-Time Service with Tolerance Range

performance in case that not all services can be run at optimal performance.

2.3.1 Timing of Non-Real-Time Services

In case of non-real-time services there is no strict expectation about the timing of a service. Although in general the expectation is that a faster timing is considered as being better, for example, by providing a more responsive human user interface. However, in case of multiple non-real-time services there might be a performance trade-off necessary between the different services, as it might be not possible to schedule all services for maximal performance. This is typically the case when these services share resources, like processing cycles, bus access, etc. For example, if one service of higher priority takes in one round significantly longer than expected, this different level of timing strictness can be used to find a best compromise for scheduling the lower-priority services.

Thus, in case of multiple non-real-time services it **also makes sense to specify their expected performance**, such that the system is able to schedule them according to their relative performance expectations. The performance metrics like throughput or latency might be specified by mean values, or by intervals of their mean values.

2.3.2 Timing of Real-Time Services

For real-time services there are concrete expectations about the boundaries of performance metrics. Most important is the latency of real-time services, which might be specified as maximum allowed latency or as a latency interval in case it is also important to specify the shortest allowed latency. Often the throughput of real-time services is important as well. The throughput might be specified by the minimum required throughput. Optionally the throughput of real-time services might be specified as an interval, in case it is important to limit the maximum processing rate as well. The allowed jitter is given by the allowed variability of the latency.

An important question about any specified performance boundaries of real-time services is how strict they are. In case that any timing violation of a service would put the system at high risk of damaging failure, the service is called a *hard real-time* service.

If a timing violation of a service can lead to a reduction of its utility, but not to any intolerable damage, then the service is called a *soft real-time* service. A soft real-time service will continue to have a positive TSUF at least for some of the timing violations. Depending on the concrete application, there is a wide range of what probability of timing violations is still acceptable.

2.4 Service Criticality

Another aspect of the temporal service utility is the categorisation of its possible negative value, defining the criticality of the service. A service with a possibly high negative utility is of high criticality. Similarly, a service with none or very low possible negative criticality is of low criticality. In between, the service might be classified of medium criticality. Our classification into low, medium, and high criticality is meant to demonstrate the concept. In real application domains there are industry standards that concretely define the set of different criticality levels and how their classification is done. For example, the DO-178B standard of the civil avionics domain defines five *Design Assurance Levels* (DAL) [10], the ISO 26262 standard of the automotive domain defines four *Automotive Safety Integrity Levels* (ASIL) [6], and the IEC 61508 standard of the automation domain defines four *Safety Integrity Levels* (SIL) [3].

2.5 Mixed Time-Criticality Systems

Based on the concepts described in this section, we can now define mixed time-criticality systems as follows:

Definition 2.1 (Mixed Time-Criticality) *is a property of systems comprising multiple services, of which at least one service is a real-time service, and at least one of the the following properties holds:*

1. *the services include different values of performance metrics or tolerance ranges,*
2. *the services include different levels of timing strictness,*
3. *the services include different criticality levels.*

Definition 2.1 states that a system with mixed-time critical services has services of different importance and/or services with different timing strictness or performance requirements. These three properties are linked together via the TSUF.

All these properties will be taken into account by the scheduler to maximise provision of required services. The mixed time-criticality information can be used to prioritise services in case the available resources are not sufficient to provide all services.

3 Specification of Mixed Time-Criticality

In Section 2 we have introduced the concept of mixed time-criticality and discussed it constituents. In the following we outline what specifications the source code of an MTC system could include in order to specify the mixed time-criticality of the system's services. We discuss this aspect here on a rather conceptual level, without focusing on a specific programming language.

Since the specification of mixed time-criticality properties has to be done for each individual service, it is natural to add the information to that software regions that represent that service.

3.1 Specification of Performance Metrics

As described in Section 2.2, the performance of a service might be described by its latency, throughput, and jitter. We propose the specification of these performance metrics by a primary specification and an optional tolerance range, as explained in the following. As described in Section 2.2.1, the so-called *tolerance-range* is meant to describe an additional operational range of reduced but still acceptable service utility. This tolerance range can give the system a means to decide at runtime in case of unexpected resource shortage.

Specification of throughput [Hz]:

`mean/minimum/range`, optional `tolerance-range`

For non-real-time services we specify the expected mean value of throughput. For real-time services we specify the required minimal value of throughput, but in case the maximum value of throughput also needs to be bounded, we specify the throughput as a range.

Specification of latency [ms]:

`mean/maximum/range`, optional `tolerance range`

For non-real-time services we specify the expected mean value of latency. For real-time services we specify the required maximal value of latency, but in case the minimal value of latency also need to be bounded, we specify the latency as a range.

Specification of jitter [ms]:

`maximum`, optional `tolerance-range`

To bound the jitter we specify the maximum jitter. Optionally, we might want to specify an additional tolerance range for the jitter beyond the primary maximum value.

The specification of all three performance metrics is not mandatory. One might only specify those for which requirements exist.

3.1.1 Specification of Timing-strictness

As described in Section 2.3, it is useful to specify how strict the given performance metrics are meant to be. In addition, in Section 3.1 we introduced two classes of performance metrics, the primary one and an optional one for reduced but still acceptable service utility. To support that we propose the following classifier of timing strictness:

Specification of Firm Deadline: `FIRMRT`

In case that no violation of the technical deadline is considered viable, the timing strictness is firm real-time. With this timing strictness a specification of a tolerance range for performance metrics is not meaningful, as the service is expected to be always within the primary performance limits.

Specification of Soft Deadline: `SOFTRT(p1)`, optional `SOFTRT-tolerance(p2)`

If a violation of the primary performance limits within a certain probability is considered viable, then we speak of soft real-time systems. The parameter $p1$ is the probability of violating the primary performance limits, and $p2$ is the probability of violating the performance limits of the optional tolerance range. This violation probability is given as violations per hour, with ultra-dependable systems like civil avionics requiring a value smaller than 10^{-9} as there the overall probability of any type of failure is limited to 10^{-9}. The optional property `SOFTRT-tolerance`($l2$) makes only sense if also any tolerance range of a performance metrics has been specified. To make use of the tolerance range it is required that $p1 > p2$.

Specification of Non-Real-Time: `NONRT`

In case a service is considered to be practically non-real-time, then it does not have any timing strictness, expressed by the attribute `NONRT`.

3.1.2 Specification of Service Criticality

As explained in Section 2.4, also the service criticality is an important decision-criterion for scheduling the processor resources in case of only insufficient resources being available to run all services at optimal performance. We propose the specification of criticality levels in a generic way using integer values:

`CRIT(p)`

with $p = 0$ representing the lowest criticality. A meaningful specification of service criticality levels has to be compliant with the domain-specific safety standards, as mentioned in Section 2.4. For example, if the safety standard uses four levels, one might use four criticality levels as a direct relation, or use even more, if the engineering approach would need a more fine-grained subdivision, e.g., to derive scheduling priorities.

4 Examples of Mixed Time-Criticality

In the following we briefly describe different applications that fit to the concept of mixed time-criticality. As a general pattern, it can be also the case that the time-criticality of a service changes dynamically, for example, by changing the mode the system is in.

4.1 Services of Different Timing-strictness but Same Criticality

There are cases where a service is of high criticality, even though the timing strictness allows for some flexibility. For example, in an aircraft there are numerous services that are essential for the safety of the passengers, even though their utilisation is not bounded by a firm deadline.

For example, the cabin pressurisation in a plane is controlled by an outflow valve to create a comfortable environment for aircraft passengers. Controlling the cabin pressurisation is of high criticality, as a failure to do so can put the passengers' life ar risk. But the service is of relatively moderate timing strictness, as its corresponding TSUF has a rather smooth transition from normal safe operation to damaging failure. Once the cabin pressure leaves the comfort zone, the passengers are still being able to compensate for a certain amount of pressure loss by hyperventilation [14]. This would give additional time to control the cabin pressure in order to avoid serious damage.

4.2 Services of Same Timing Strictness but Different Criticality

As a generic pattern, using a time-triggered (TT) communication interface enforces technical deadline with high timing strictness, imposed by the length of the so-called TT communication round. Any two services running on a node would both be required to adhere to the same timing strictness to function flawlessly. However, these two services might have different criticality levels. Based on these different criticality levels the local scheduler of a node can prefer the execution of the higher criticality service, allowing for a higher probability to maintain its service in case of unexpected increases of processing latency.

4.3 Services of Different Timing Strictness and also Different Criticality

Schrijver and Creemers from Philips Healthcare published a use case of X-ray treatment with image processing [12]. This use case has basically three services to provide: real-time image filtering (IF), real-time feature detection (FD), non-real-time post-processing (PP). The IF service is meant to provide a frame rate of 24 frames/second. The FD service is significantly more complex to calculate than IF, but at the same time has a lower update rate requirement than IF. The update rate of FD depends on the available free computing resources and should ideally cope with the moving speed of the relevant object to be highlighted in the video stream. The PP service is meant to run in the background when none of the IF and FD services is active. Without running the PP in background and instead executing them just at the end of the day when switching off the machine, would result in a considerable delay of the shutdown process. The three services also have different criticality levels, with IF having the highest criticality, and PP the lowest criticality.

5 Summary and Conclusion

In this paper we presented a generic definition of mixed time-criticality systems. Important for this definition is that it is not motivated by any specific scheduling method, but rather from the application engineering point of view. Central for the development of the generic concept of mixed time-criticality has been the uniform characterisation of hard-real-time, soft-real-time, and non-real-time services by their so-called *temporal service utility function* (TSUF). The TSUF expresses the criticality as well as the timing requirements of services. Also important is the concept of tolerance ranges, which provides a more flexible concept for fault-tolerant scheduling than using criticality levels alone.

We further presented attributes for programming models in order to specify services of mixed time-criticality. We also presented some applications that can be modelled with mixed time-criticality.

Our future work is to derive a concrete programming model for the presented definition of mixed time-criticality. Furthermore, we will use the concept of mixed time-criticality to set the frame for corresponding scheduling research.

Acknowledgments

The author would like to thank Michael Zolda for useful comments on earlier versions of this text.

References

[1] S. Baruah, A. Burns, and R. Davis. Response-time analysis for mixed criticality systems. In *Proc. 32nd Real-Time Systems Symposium (RTSS)*, pages 34–43. IEEE, 2011.

[2] S. Baruah, B. Chattopadhyay, H. Li, and I. Shin. Mixed-criticality scheduling on multiprocessors. *Real-Time Systems*, pages 1–36, 2013.

[3] I. E. Commission. Functional safety of electrical / electronic / programmable electronic safety-related systems. IEC standard 61508, 1998.

[4] K. Goossens, A. Azevedo, K. Chandrasekar, M. D. Gomony, S. Goossens, M. Koedam, Y. Li, D. Mirzoyan, A. Molnos, A. B. Nejad, A. Nelson, and S. Sinha. Virtual execution platforms for mixed-time-criticality applications: the CompSoC architecture and design flow. In editor, editor, *Proc. 5th Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, pages 23–30, San Juan, Puerto Rico, Dec. 2012.

[5] D. Goswami, M. Lukasiewycz, R. Schneider, and S. Chakraborty. Time-triggered implementations of mixed-criticality automotive software. In *Proc. Conference on Design, Automation and Test in Europe*, pages 1227–1232, Los Alamitos, CA, USA, 2012. IEEE Computer Society.

[6] ISO/DIS. Road vehicles – functional safety. ISO/DIS standard 26262, Nov 2011. International Standard.

[7] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Proc. IFIP Congress on Information Processing*, Stockholm, Sweden, Aug. 1974. ISBN: 0-7204-2803-3.

[8] H. Kopetz. *Real-Time Systems - Design Principles for Distributed Embedded Applications*. Springer, 2nd edition, 2011. ISBN: 978-1-4419-8236-0.

[9] H. Li and S. Baruah. Outstanding paper award: Global mixed-criticality scheduling on multiprocessors. In *Proc. 24th Euromicro Conference on Real-Time Systems*, pages 166–175, Los Alamitos, CA, USA, 2012. IEEE Computer Society.

[10] RTCA. Software considerations in airborne systems and equipment certification. RTCA/DO-178B, 1992.

[11] C. E. Salloum, M. Elshuber, O. Höftberger, H. Isakovic, and A. Wasicek. The across mpsoc - a new generation of multi-core processors designed for safety-critical embedded systems. In *Proc. 5th Euromicro Conference on Digital System Design (DSD)*, pages 105–113, Cesme, Izmir, Turkey, Sep. 2012. IEEE.

[12] M. Schrijver and M. Creemers. Running real-time and best-effort applications concurrently on common off-the-shelf hardware. In C. Grelck, K. Hammond, and S. Scholz, editors, *2nd HiPEAC Workshop on Feedback-Directed Compiler Optimization for Multicore Architectures (FD-COMA'13)*, Berlin, Germany, Jan. 2013. HiPEAC.

[13] D. Tamas-Selicean and P. Pop. Design optimization of mixed-criticality real-time applications on cost-constrained partitioned architectures. In *Proc. 32nd Real-Time Systems Symposium (RTSS)*, pages 24–33. IEEE, 2011.

[14] Wikipedia. Cabin pressurization. web page: `http://en.wikipedia.org/wiki/Cabin_pressurization`, 2013. accessed online on 13th August 2013 at 21:00.

Turning Conjectures into Positive Knowledge: Proving Precision of Worst-Case Execution-Time Bounds

Jens Knopp
TU Wien

Laura Kovacs
Chalmers Univ. of Technology

Jakob Zwirchmayr
TU Wien

Zusammenfassung

Der Vortrag wurde von Jens Knopp auf dem Kolloqium gehalten.

Extending a Model-Driven Approach for the Cross-Platform Development of Business Apps

Henning Heitkötter Herbert Kuchen
Tim A. Majchrzak

Universität Münster, Institut für Wirtschaftsinformatik
{heitkoetter,kuchen,tima}@wi.uni-muenster.de

1 Extended Abstract

Due to the heterogeneity of different platforms, it is an expensive endeavor to provide a mobile application (app) for several of them. Cross-platform development approaches can solve this problem. Existing cross-platform approaches [HTM13, Apa13, App13] have severe limitations and typically work on a low-level of abstraction. Our model-driven [SV06] cross-platform approach MD^2 focuses on the domain of business apps and, hence, reaches a high-level of abstraction while maintaining a platform-specific look & feel. A textual model written in an MVC-based DSL is automatically transformed into native apps for Android and iOS [HMWK12, HMK13].

In contrast to other approaches, our approach reaches a high level of abstraction and hence offers high productivity while avoiding performance penalties. In the full paper [HMK14], we focus on extensions of MD^2. First, we describe how to add recursion to the DSL, which allows us to reach Turing completeness also for offline computations. Moreover, we investigated means of using the nesting structure of containers in the user interface for a device-specific layout. In the future, we would like to broaden the scope of MD^2 further. Another area for future work is the inclusion of individual, manually written code, for example, using the Generation Gap pattern [Fow11, pp. 571ff.].

References

[Apa13] Apache cordova, 2013. http://cordova.apache.org/.

[App13] Appcelerator, 2013. http://www.appcelerator.com/.

[Fow11] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley Pearson Education, 2011.

[HMK13] Henning Heitkötter, Tim A. Majchrzak, and Herbert Kuchen. Cross-platform model-driven development of mobile applications with MD^2. In *Proc. SAC '13*, pages 526–533. ACM, 2013.

[HMK14] Henning Heitkötter, Tim A. Majchrzak, and Herbert Kuchen. Extending a model-driven cross-platform development approach for business apps. *to appear*, 2014.

[HMWK12] Henning Heitkötter, Tim A. Majchrzak, Ulrich Wolffgang, and Herbert Kuchen. *Business Apps: Grundlagen und Status quo*. Number 4. Förderkreis der Angewandten Informatik an der WWU Münster e.V., 2012.

[HTM13] Html5, 2013. http://www.w3.org/TR/html5/.

[SV06] Thomas Stahl and M. Völter. *Model-driven software development*. John Wiley & Sons New York, 2006.

A surprising link between functional programming and topology

(Extended Abstract)

Gunther Schmidt

Fakultät für Informatik, Universität der Bundeswehr München
85577 Neubiberg, Germany
gunther.schmidt@unibw.de

Abstract. Developing the relational language TituRel and applying it also to topological concepts revealed a surprising link. Normally, structures are compared using homomorphisms and sometimes isomorphisms. This applies to group homomorphisms, to graph homomorphisms and many more. The technique of comparison for topological structures is quite different. Lifting concepts to a relational and, thus, algebraically manipulable and shorthand form shows that existential and inverse images must be used — well-known from the area of functional programming.

Keywords relational mathematics, homomorphism, topology, continuity.

1 Prerequisites

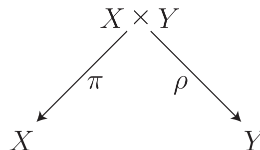
We will work with heterogeneous relations and provide a general reference to [Sch11a], but also to the earlier [SS89,SS93]. Our operations are, thus, binary union "$\cup$", intersection "$\cap$", composition "$;$", unary negation "$\overline{}$", transposition or conversion "$^\mathsf{T}$", together with zero-ary null relations "$\mathbb{⊥}$", universal relations "$\mathbb{T}$", and identities "$\mathbb{I}$". As an often used term, we introduce the *symmetric quotient* of two relations with common source as $\mathtt{syq}(R,S) := \overline{R^\mathsf{T};\overline{S}} \cap \overline{\overline{R}^\mathsf{T};S}$, by which "column comparison" for boolean matrices becomes possible.

The language TituRel (see [Sch03,Sch11b]) also allows to use constructs that are only characterized *up to isomorphism*. These include the

• **direct product**, namely **any** pair π, ρ of relations with common source satisfying

$$\pi^\mathsf{T};\pi = \mathbb{I}, \quad \rho^\mathsf{T};\rho = \mathbb{I}, \quad \pi;\pi^\mathsf{T} \cap \rho;\rho^\mathsf{T} = \mathbb{I}, \quad \pi^\mathsf{T};\rho = \mathbb{T}.$$

`DirPro x y`	$X \times Y$
`Pi x y`	$\pi : X \times Y \longrightarrow X$
`Rho x y`	$\rho : X \times Y \longrightarrow Y$



- **direct sum**, namely **any** pair ι, κ of relations with common target satisfying

$$\iota;\iota^\mathsf{T} = \mathbb{I}, \quad \kappa;\kappa^\mathsf{T} = \mathbb{I}, \quad \iota^\mathsf{T};\iota \cup \kappa^\mathsf{T};\kappa = \mathbb{I}, \quad \iota;\kappa^\mathsf{T} = \mathbb{⫫}.$$

```
DirSum x y
Iota   x y
Kappa  x y
```

$X + Y$

$\iota : X \longrightarrow X + Y$

$\kappa : Y \longrightarrow X + Y$

X —ι→ $X+Y$ ←κ— Y

- **direct power**, namely **any** relation ε satisfying

$$\mathtt{syq}(\varepsilon, \varepsilon) \subseteq \mathbb{I}, \quad \mathtt{syq}(\varepsilon, R) \text{ is surjective for every relation } R \text{ starting in } X.$$

```
DirPow x
Member x
```

$\mathbf{2}^X$

$\varepsilon : X \longrightarrow \mathbf{2}^X$

X —ε→ $\mathcal{P}(X)$; X —R→ Y; $\mathcal{P}(X)$ —$\mathtt{syq}(\varepsilon,R)$→ Y

One will observe the 1st-order style of the definition in contrast to the equational style of the product definition, both obtained using relations. Classically, the characterisation of the powerset requires 2nd-order and of the product 1st-order.

One will also observe the fractal structure in the following example of a membership relation, together with an interesting interplay between subsets, their ordering $\Omega := \overline{\varepsilon^\mathsf{T};\overline{\varepsilon}}$, and elements like e in the powerset.

$U = \varepsilon;e \qquad e = \mathtt{syq}(\varepsilon, U)$

	{}	{a}	{b}	{a,b}	{c}	{a,c}	{b,c}	{a,b,c}	{d}	{a,d}	{b,d}	{a,b,d}	{c,d}	{a,c,d}	{b,c,d}	{a,b,c,d}	U
a	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
b	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	1
c	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0
d	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1

$(0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0) = e^\mathsf{T}$

	{}	{a}	{b}	{a,b}	{c}	{a,c}	{b,c}	{a,b,c}	{d}	{a,d}	{b,d}	{a,b,d}	{c,d}	{a,c,d}	{b,c,d}	{a,b,c,d}	e
{}	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
{a}	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
{b}	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
{a,b}	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0
{c}	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0
{a,c}	0	0	0	0	0	1	0	1	0	0	0	0	0	1	0	1	0
{b,c}	0	0	0	0	0	0	1	1	0	0	0	0	0	0	1	1	0
{a,b,c}	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0
{d}	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0
{a,d}	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	1	0
{b,d}	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	1	1
{a,b,d}	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0
{c,d}	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0
{a,c,d}	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0
{b,c,d}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0
{a,b,c,d}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

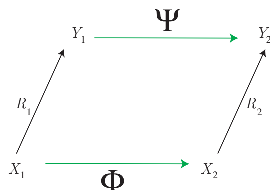
Subset U and corresponding point e in the powerset via ε, Ω

Along with the direct product, we automatically have the Kronecker product and (when sources coincide) the fork operator for relations,

$$(R \otimes S) := \pi;R;\pi'^\mathsf{T} \cap \rho;S;\rho'^\mathsf{T} \quad \text{and} \quad (P \otimes Q) := P;\pi'^\mathsf{T} \cap Q;\rho'^\mathsf{T}.$$

2 Comparing structures

Comparison of structures via homomorphisms or structure-preserving mappings is omnipresent in mathematics, be it for groups, lattices, modules, graphs, or others. Most of these follow a general schema.



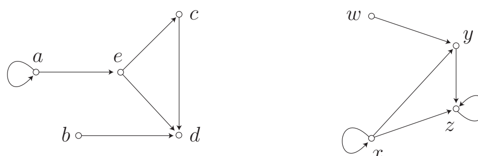
Two “structures” of whatever kind are given by a relation $R_1 : X_1 \longrightarrow Y_1$ and a relation $R_2 : X_2 \longrightarrow Y_2$. With mappings $\Phi : X_1 \longrightarrow X_2$ and $\Psi : Y_1 \longrightarrow Y_2$ they shall be compared, and we may ask whether these mappings transfer the first structure “sufficiently nice” into the second one.

2.1 Definition. Φ, Ψ is a **homomorphism** from R_1 to R_2, if $R_1 ; \Psi \subseteq \Phi ; R_2$. The two Φ, Ψ constitute an **isomorphism**, if Φ, Ψ as well as $\Phi^\mathsf{T}, \Psi^\mathsf{T}$ are homomorphisms. □

If any two elements x, y are related by R_1, so are their images $\varphi(x), \psi(y)$ by R_2:

$$\forall x \in X_1 : \forall y \in Y_1 : (x, y) \in R_1 \rightarrow (\varphi(x), \psi(y)) \in R_2.$$

This concept is also suitable for relational structures; it works in particular for a graph homomorphism Φ, Φ as in the example:



$$R_1 = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{pmatrix} \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0} \end{pmatrix} \end{array} \qquad \Phi = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{cccc} w & x & y & z \end{array} \\ \begin{pmatrix} \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \end{pmatrix} \end{array} \qquad R_2 = \begin{array}{c} \\ w \\ x \\ y \\ z \end{array} \begin{array}{c} \begin{array}{cccc} w & x & y & z \end{array} \\ \begin{pmatrix} \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \end{pmatrix} \end{array}$$

We recall the rolling of homomorphisms:

2.2 Theorem. If Φ, Ψ are mappings, then

$$R_1 ; \Psi \subseteq \Phi ; R_2 \iff R_1 \subseteq \Phi ; R_2 ; \Psi^{\mathsf{T}} \iff \Phi^{\mathsf{T}} ; R_1 \subseteq R_2 ; \Psi^{\mathsf{T}} \iff \Phi^{\mathsf{T}} ; R_1 ; \Psi \subseteq R_2$$

If relations Φ, Ψ are not mappings, one cannot fully execute this rolling; there remain different forms of (bi-)simulations as explicated in [dRE98].

3 Recalling concepts of topology

Now, we try to apply this well-known technique for topological structures. Topology may be defined via open or closed sets, neighborhoods, transition to open kernels, etc. We show that at least the neighborhood version — in the form given by Felix Hausdorff — shows an inherently "linear" configuration, that it is apt to being formulated using relations.

3.1 Definition. A set X endowed with a system $\mathcal{U}(p)$ of subsets — called neighborhoods — for every $p \in X$ is called a **topological structure**, provided

i) $p \in U$ for every neighborhood $U \in \mathcal{U}(p)$
ii) If $U \in \mathcal{U}(p)$ and $V \supseteq U$, then $V \in \mathcal{U}(p)$
iii) If $U_1, U_2 \in \mathcal{U}(p)$, then $U_1 \cap U_2 \in \mathcal{U}(p)$ and $X \in \mathcal{U}(p)$
iv) For every $U \in \mathcal{U}(p)$ there is a $V \in \mathcal{U}(p)$ so that $U \in \mathcal{U}(y)$ for all $y \in V$. □

The same is now expressed with membership with ε, conceiving $\mathcal{U}$ as a relation, and gradually lifted to a relational form:

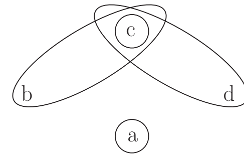
$$\varepsilon : X \longrightarrow \mathbf{2}^X \quad \text{and} \quad \mathcal{U} : X \longrightarrow \mathbf{2}^X$$

"For every $U \in \mathcal{U}(p)$ there exists a $V \in \mathcal{U}(p)$ such that $U \in \mathcal{U}(y)$ for all $y \in V$"

$$\begin{aligned}
&\forall p, U : U \in \mathcal{U}(p) \;\rightarrow\; \big(\exists V : V \in \mathcal{U}(p) \wedge \big(\forall y : y \in V \rightarrow U \in \mathcal{U}(y)\big)\big)\\
&\forall p, U : \mathcal{U}_{pU} \;\rightarrow\; \big(\exists V : \mathcal{U}_{pV} \wedge \big(\forall y : \varepsilon_{yV} \rightarrow \mathcal{U}_{yU}\big)\big)\\
&\forall p, U : \mathcal{U}_{pU} \;\rightarrow\; \big(\exists V : \mathcal{U}_{pV} \wedge \overline{\exists y : \varepsilon_{yV} \wedge \overline{\mathcal{U}_{yU}}}\big)\\
&\forall p, U : \mathcal{U}_{pU} \;\rightarrow\; \big(\exists V : \mathcal{U}_{pV} \wedge \overline{\varepsilon^{\mathsf{T}} ; \overline{\mathcal{U}}}_{VU}\big)\\
&\forall p, U : \mathcal{U}_{pU} \;\rightarrow\; \big(\mathcal{U} ; \overline{\varepsilon^{\mathsf{T}} ; \overline{\mathcal{U}}}\big)_{pU}\\
&\mathcal{U} \subseteq \mathcal{U} ; \overline{\varepsilon^{\mathsf{T}} ; \overline{\mathcal{U}}}
\end{aligned}$$

An example of a neighborhood topology and the basis of its open sets:

$$\mathcal{U} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array}\!\!\begin{array}{cccccccccccccccc} \{\} & \{a\} & \{b\} & \{a,b\} & \{c\} & \{a,c\} & \{b,c\} & \{a,b,c\} & \{d\} & \{a,d\} & \{b,d\} & \{a,b,d\} & \{c,d\} & \{a,c,d\} & \{b,c,d\} & \{a,b,c,d\} \\ 0&1&0&1&0&1&0&1&0&1&0&1&0&1&0&1\\ 0&0&0&0&0&0&1&1&0&0&0&0&0&0&1&1\\ 0&0&0&0&1&1&1&1&0&0&0&0&1&1&1&1\\ 0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1 \end{array}$$



This, together with a transfer of the other properties to the relational level, gives rise to lift the whole Definition 3.1, thus making it point-free.

3.2 Definition. A relation $\mathcal{U} : X \longrightarrow \mathbf{2}^X$ will be called a **neighborhood topology** if the following properties are satisfied:

i) $\mathcal{U};\mathbb{T} = \mathbb{T}$ and $\mathcal{U} \subseteq \varepsilon$,
ii) $\mathcal{U};\Omega \subseteq \mathcal{U}$,
iii) $(\mathcal{U} \otimes \mathcal{U});\mathcal{M} \subseteq \mathcal{U}$,
iv) $\mathcal{U} \subseteq \mathcal{U};\overline{\varepsilon^{\mathsf{T}};\overline{\mathcal{U}}}$. □

Correspondingly, this may be executed for the various other topology concepts:

3.3 Definition. We call a relation $\mathcal{K} : \mathbf{2}^X \longrightarrow \mathbf{2}^X$ a **mapping-to-open-kernel-topology**, if

i) $\mathcal{K}$ is a kernel forming, i.e., $\mathcal{K} \subseteq \Omega^{\mathsf{T}}$, $\Omega;\mathcal{K} \subseteq \mathcal{K};\Omega$, $\mathcal{K};\mathcal{K} \subseteq \mathcal{K}$,
ii) $\varepsilon;\mathcal{K}^{\mathsf{T}}$ is total,
iii) $(\mathcal{K} \otimes \mathcal{K})\mathcal{M} \subseteq \mathcal{M};\mathcal{K};\Omega^{\mathsf{T}}$, in fact $(\mathcal{K} \otimes \mathcal{K})\mathcal{M} = \mathcal{M};\mathcal{K}$. □

In this and the following definition, we employ the lifted binary meet $\mathcal{M}$, corresponding to set intersection.

3.4 Definition. A vector $\mathcal{O}_V$ along $\mathbf{2}^X$ will be called an **open set topology** provided

i) $\mathtt{syq}(\varepsilon, \mathbb{⊥}) \subseteq \mathcal{O}_V$ $\mathtt{syq}(\varepsilon, \mathbb{T}) \subseteq \mathcal{O}_V$
ii) $v \subseteq \mathcal{O}_V \implies \mathtt{syq}(\varepsilon, \varepsilon;v) \subseteq \mathcal{O}_V$ for all vectors $v \subseteq \mathbf{2}^X$
iii) $\mathcal{M}^{\mathsf{T}};(\mathcal{O}_V \otimes \mathcal{O}_V) \subseteq \mathcal{O}_V$ □

All these topology concepts are cryptomorphic — as could be expected. The transitions below may be written down in TituRel so as to achieve the transition intended.

$$\mathcal{U} \mapsto \mathcal{K} := \mathtt{syq}(\mathcal{U}, \varepsilon) : \mathbf{2}^X \longrightarrow \mathbf{2}^X$$

$$\mathcal{K} \mapsto \mathcal{U} := \varepsilon;\mathcal{K}^{\mathsf{T}} : X \longrightarrow \mathbf{2}^X.$$

$$\mathcal{O}_D \mapsto \mathcal{U} := \varepsilon;\mathcal{O}_D;\Omega$$

$$\mathcal{O}_D \mapsto \mathcal{O}_V := \mathcal{O}_D;\mathbb{T}$$

$$\mathcal{K}, \mathcal{U}, \mathcal{O}_V \quad \mapsto \quad \mathcal{O}_D := \mathbb{I} \cap \overline{\varepsilon^{\mathsf{T}}; \overline{\mathcal{U}}} = \mathbb{I} \cap \mathcal{O}_V; \mathbb{T} = \mathcal{K}^{\mathsf{T}}; \mathcal{K}$$

One may, thus, obtain the same topology in different forms.

ε

	{}	{a}	{b}	{a,b}	{c}	{a,c}	{b,c}	{a,b,c}	{d}	{a,d}	{b,d}	{a,b,d}	{c,d}	{a,c,d}	{b,c,d}	{a,b,c,d}
a	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
b	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
c	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
d	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

$\mathcal{U}$

	{}	{a}	{b}	{a,b}	{c}	{a,c}	{b,c}	{a,b,c}	{d}	{a,d}	{b,d}	{a,b,d}	{c,d}	{a,c,d}	{b,c,d}	{a,b,c,d}
a	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
b	0	0	0	0	0	0	1	1	0	0	0	0	0	0	1	1
c	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
d	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1

$\varepsilon_{\mathcal{O}} := \varepsilon \cap \mathbb{T}; \mathcal{O}_V^{\mathsf{T}} = \varepsilon; \mathcal{K} \cap \varepsilon$

	{}	{a}	{b}	{a,b}	{c}	{a,c}	{b,c}	{a,b,c}	{d}	{a,d}	{b,d}	{a,b,d}	{c,d}	{a,c,d}	{b,c,d}	{a,b,c,d}
a	0	1	0	0	0	1	0	1	0	0	0	0	0	1	0	1
b	0	0	0	0	0	0	1	1	0	0	0	0	0	0	1	1
c	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
d	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1

$\mathcal{K} := \mathtt{syq}(\mathcal{U}, \varepsilon)$ indicating $\mathcal{O}_D$ as diagonal

	{}	{a}	{b}	{a,b}	{c}	{a,c}	{b,c}	{a,b,c}	{d}	{a,d}	{b,d}	{a,b,d}	{c,d}	{a,c,d}	{b,c,d}	{a,b,c,d}
{}	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
{a}	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
{b}	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
{a,b}	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
{c}	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
{a,c}	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
{b,c}	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
{a,b,c}	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
{d}	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
{a,d}	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
{b,d}	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
{a,b,d}	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
{c,d}	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
{a,c,d}	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
{b,c,d}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
{a,b,c,d}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Open sets as membership and open kernel mapping

By the way, there exists also a kernel-forming that doesn't lead to a topology; it is not intersection-closed as can be seen from the subsets $\{a,b\}$ and $\{b,d\}$ with intersection $\{d\}$:

$\mathcal{K} =$

	{}	{a}	{b}	{a,b}	{c}	{a,c}	{b,c}	{a,b,c}	{d}	{a,d}	{b,d}	{a,b,d}	{c,d}	{a,c,d}	{b,c,d}	{a,b,c,d}
{}	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
{a}	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
{b}	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
{a,b}	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
{c}	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
{a,c}	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
{b,c}	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
{a,b,c}	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
{d}	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
{a,d}	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
{b,d}	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
{a,b,d}	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
{c,d}	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
{a,c,d}	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
{b,c,d}	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
{a,b,c,d}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

One might naively be tempted to study also the comparison of topologies with the concept of homomorphism. For that, we recall the standard definition of continuity.

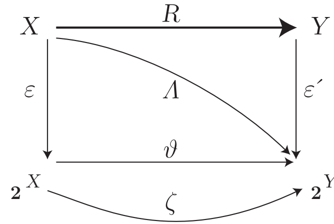
3.5 Definition. Let any two neighborhood topologies $\mathcal{U},\mathcal{U}'$ be given on sets X, X', and consider a mapping $f : X \longrightarrow X'$. One calls

f **continuous** $:\iff$ For every point $p \in X$ and every $U' \in \mathcal{U}'(f(p))$, there exists a $U \in \mathcal{U}(p)$ such that $f(U) \subseteq U'$.

When lifting this definition to the point-free level, we will soon learn that homomorphisms do not work properly; instead, we have to employ the concept of an existential image and an inverse image.

4 Existential and inverse image

The lifting of a relation R to a corresponding relation ϑ_R on the powerset level has been called its existential image; cf. [Bd96]. (There exists also the power transpose Λ_R and the power relator ζ_R.)



4.1 Definition. Assume an arbitrary relation $R : X \longrightarrow Y$ with membership relations $\varepsilon : X \longrightarrow \mathbf{2}^X$ and $\varepsilon' : Y \longrightarrow \mathbf{2}^Y$ on either side and define

$\vartheta := \vartheta_R := \mathtt{syq}(R^\mathsf{T};\varepsilon, \varepsilon') = \overline{\varepsilon^\mathsf{T};R;\overline{\varepsilon'}} \cap \overline{\overline{\varepsilon^\mathsf{T};R};\varepsilon'}$, its **existential image**.

The **inverse image** is obtained when taking the existential image of the transposed relation. □

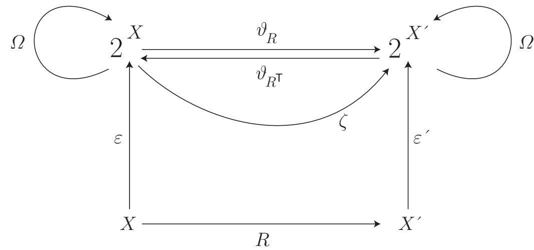
It turns out that ϑ is

- (lattice-)continuous mapping wrt. the powerset orders $\Omega = \overline{\varepsilon^\mathsf{T};\overline{\varepsilon}}$,
- multiplicative: $\vartheta_{Q;R} = \vartheta_Q;\vartheta_R$,
- preserves identities: $\vartheta_{\mathbb{I}_X} = \mathbb{I}_{\mathbf{2}^X}$,
- R may be re-obtained from ϑ_R as $R = \overline{\varepsilon;\vartheta_R;\overline{\varepsilon'}^\mathsf{T}}$.

It also satisfies a the following simulation property.

4.2 Theorem. R and its existential image as well as its inverse image **simulate** each other via $\varepsilon, \varepsilon'$:

$$\varepsilon^{\mathsf{T}};R = \vartheta_R;\varepsilon'^{\mathsf{T}} \qquad \varepsilon'^{\mathsf{T}};R^{\mathsf{T}} = \vartheta_{R^{\mathsf{T}}};\varepsilon^{\mathsf{T}}. \qquad \square$$



$R =$

	a	b	c	d
1	0	1	0	1
2	1	0	0	0
3	0	0	1	0
4	0	0	0	1
5	1	0	1	0

$\vartheta_R =$

	{}	{a}	{b}	{a,b}	{c}	{a,c}	{b,c}	{a,b,c}	{d}	{a,d}	{b,d}	{a,b,d}	{c,d}	{a,c,d}	{b,c,d}	{a,b,c,d}
{}	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
{1}	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
{2}	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
{1,2}	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
{3}	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
{1,3}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
{2,3}	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
{1,2,3}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
{4}	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
{1,4}	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
{2,4}	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
{1,2,4}	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
{3,4}	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
{1,3,4}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
{2,3,4}	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
{1,2,3,4}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
{5}	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
{1,5}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
{2,5}	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
{1,2,5}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
{3,5}	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
{1,3,5}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
{2,3,5}	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
{1,2,3,5}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
{4,5}	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
{1,4,5}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
{2,4,5}	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
{1,2,4,5}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
{3,4,5}	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
{1,3,4,5}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
{2,3,4,5}	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
{1,2,3,4,5}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

The inverse image is obviously not the transpose of the existential image.

$\vartheta_{R^\mathsf{T}} =$

	{}	{1}	{2}	{1,2}	{3}	{1,3}	{2,3}	{1,2,3}	{4}	{1,4}	{2,4}	{1,2,4}	{3,4}	{1,3,4}	{2,3,4}	{1,2,3,4}	{5}	{1,5}	{2,5}	{1,2,5}	{3,5}	{1,3,5}	{2,3,5}	{1,2,3,5}	{4,5}	{1,4,5}	{2,4,5}	{1,2,4,5}	{3,4,5}	{1,3,4,5}	{2,3,4,5}	{1,2,3,4,5}
{}	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
{a}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
{b}	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
{a,b}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
{c}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
{a,c}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
{b,c}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
{a,b,c}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
{d}	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
{a,d}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
{b,d}	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
{a,b,d}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
{c,d}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
{a,c,d}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
{b,c,d}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
{a,b,c,d}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

5 Expressing continuity via the inverse image

We will now use the inverse image to formulate continuity.

$$\begin{array}{ccc} 2^X & \xleftarrow{\;\vartheta_{f^\mathsf{T}}\;} & 2^{X'} \\ \uparrow \mathcal{U} & & \uparrow \mathcal{U}' \\ X & \xrightarrow{\;f\;} & X' \end{array}$$

First, continuity shall be lifted so as to arrive at a relational form.

5.1 Definition. Consider two neighborhood topologies $\mathcal{U} : X \longrightarrow \mathbf{2}^X$ and $\mathcal{U}' : X' \longrightarrow \mathbf{2}^{X'}$ as well as a mapping $f : X \longrightarrow X'$. We call

$$f \text{ (neighborhood-)}\textbf{continuous} \quad :\Longleftrightarrow \quad f{;}\mathcal{U}'{;}\vartheta_{f^\mathsf{T}} \subseteq \mathcal{U} \qquad \square$$

The equivalent version $f{;}\mathcal{U}' \subseteq \mathcal{U}{;}\vartheta^\mathsf{T}_{f^\mathsf{T}}$ is obtained by shunting the mapping ϑ_{f^T}. One cannot do so with regard to the mapping f.

5.2 Definition. Given sets X, X' with topologies, we consider a mapping $f : X \longrightarrow X'$ together with its inverse image $\vartheta_{f^\mathsf{T}} : \mathbf{2}^{X'} \longrightarrow \mathbf{2}^X$. Then we say that the pair $(f, \vartheta_{f^\mathsf{T}})$ is

i) (open-kernel-map-)**continuous** $:\iff \mathcal{K}_2^{\mathsf{T}};\vartheta_{f^{\mathsf{T}}} \subseteq \overline{{\varepsilon_2}^{\mathsf{T}};f^{\mathsf{T}};\overline{\varepsilon_1}};\mathcal{K}_1^{\mathsf{T}}$

ii) (open-diagonal-)**continuous** $:\iff \mathcal{O}_{D2};\vartheta_{f^{\mathsf{T}}} \subseteq \vartheta_{f^{\mathsf{T}}};\mathcal{O}_{D1}$

iii) (open-set-)**continuous** $:\iff \vartheta^{\mathsf{T}}_{f^{\mathsf{T}}};\mathcal{O}'_V \subseteq \mathcal{O}_V$

iv) (membership-in-open-sets-)**continuous** $:\iff f;\varepsilon_{\mathcal{O}_2};\vartheta_{f^{\mathsf{T}}} \subseteq \varepsilon_{\mathcal{O}_1}$ □

All these versions of continuity can be shown to be equivalent.

Literatur

Bd96. Richard S. Bird and Oege de Moor. *Algebra of Programming*. Prentice-Hall International, 1996.

dRE98. Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Number 47 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1998.

Sch03. Gunther Schmidt. Relational Language. Technical Report 2003-05, Fakultät für Informatik, Universität der Bundeswehr München, 2003. 101 pages, `http://mucob.dyndns.org:30531/~gs/Papers/LanguageProposal.html`.

Sch11a. Gunther Schmidt. *Relational Mathematics*, volume 132 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 2011. ISBN 978-0-521-76268-7, 584 pages.

Sch11b. Gunther Schmidt. TituRel: Sprache für die Relationale Mathematik. Technical Report 132, Arbeitsberichte des Instituts für Wirtschaftsinformatik, Universität Münster, 2011. 11 pages, `http://mucob.dyndns.org:30531/~gs/Papers/Raesfeld2011ExtendedAbstract.pdf`.

SS89. Gunther Schmidt and Thomas Ströhlein. *Relationen und Graphen*. Mathematik für Informatiker. Springer-Verlag, 1989. ISBN 3-540-50304-8, ISBN 0-387-50304-8.

SS93. Gunther Schmidt and Thomas Ströhlein. *Relations and Graphs — Discrete Mathematics for Computer Scientists*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1993. ISBN 3-540-56254-0, ISBN 0-387-56254-0.

Finding Security Bugs in Java Programs using Datalog

Bernhard Scholz

Oracle Labs
300 Ann Street
Brisbane
QLD4000, Australia

Zusammenfassung

Recently, various zero-day exploits emerged for Java making computers that run Java potentially vulnerable. Though Java was designed with a strong emphasis on security and the language itself is type-safe, defects in the Java JDK library permit attackers to break the security of Java.

This talk gives an overview of the activities at Oracle Labs that has been developing a program analysis tool for Java. The program analysis tool will be able to identify and report security defects in the JDK library. In a pilot project, we specify security defects of Java programs in a restricted variant of Horn-Logic called Datalog. The declarative approach of expressing static program analyses has various advantages.

Implementierung eines Typinferenzalgorithmus für Java 8

Andreas Stadelmeier und Martin Plümicke

24. August 2013

Abstract

Java 8, die nächste anstehende Erweiterung der Programmiersprache Java, führt neue Features in der Sprache ein. Dabei handelt es sich unter anderem um sogenannte Lambda-Ausdrücke, welche das Ablegen von Funktionen in Variablen ermöglichen und dadurch Elemente der funktionalen Programmierung in Java einführen.
Dieser Artikel beschreibt die Implementierung eines zu Java 8 kompatiblen Typinferenzalgorithmus. Ziel ist die Ermöglichung von automatisierter Typberechnung in Java mithilfe von Typinferenz. Standardmäßig ist Java streng typisiert. Jeder Variablen muss bei der Deklaration ein Typ zugewiesen werden. Ein Typinferenzalgorithmus kann diesen Arbeitsschritt jedoch übernehmen, indem er den Typ einer Variablen anhand ihrer Verwendung ermittelt und einsetzt.

1 Einleitung

1.1 Problemstellung

In der statisch typisierten Sprache Java müssen die Typen sämtlicher Variablen und Ausdrücke bei deren Dekleration festgelegt werden. Dabei lässt sich auf deren Datentypen oft auch anhand ihres Kontexts schließen. Knapp 20 Studenten entwickelten im Zuge mehrerer Studienarbeiten (z.B. [Bäu05, Hol06, Lüd07]) einen Java-Compiler, welcher in der Lage war selbständig Typen für Java 7 zu inferieren.
Mit Java 8 erhält die Programmiersprache neue Features, welche erstmalig Elemente der funktionalen Programmierung in Java einführen. Diese sollen unter anderem die Verwendung von inneren Klassen teilweise ablösen. Sogenannte Lambda Ausdrücke, eine kompakte Form der Methodendeklaration, ermöglichen zukünftig die Übergabe von Funktionen als Parameter bei Methodenaufrufen oder das Speichern von Funktionen in Variablen. Allerdings unterstützt der ursprünglich implementierte Typinferenzalgorithmus die neuen Sprachkonstrukte nicht.

1.2 Aufgabenstellung

Ziel dieser Arbeit ist die Anpassung des in Kapitel 1.1 vorgestellten Java-Compilers an die neuen Features in Java 8 und die Ersetzung dessen Typinferenzalgorithmus. Dieser Compiler ist auf die in Java 7 verwendbaren Datentypen ausgelegt. Weder der Compiler noch sein Typinferenzalgorithmus sind in der Lage Lambda Ausdrücke zu verarbeiten. Der erste Schritt ist es daher den Compiler an die mit Java 8 eingeführten Änderungen anzupassen. Anschließend kann die überarbeitete und zu Lambda Ausdrücken kompatible Spezifikation des Typinferenzalgorithmus implementiert werden.

1.3 Ziel

Ziel dieser Arbeit ist die Umsetzung eines zu Java 8 kompatiblen Compilers mit dem in [Plüar] spezifizierten Typinferenzalgorithmus. Der Aufbau der Arbeit teilt sich in vier Teile ein. Im folgenden Kapitel 2 sind notwendigen Grundlagen und Informationen hauptsächlich zu den Neuerungen in Java 8 erklärt. In Kapitel 3 die Typen von Lambda Ausdrücken näher betrachtet. Die Analyse des bereits vorhandenen Java-Compilers und der Datentypen, welche in dieser Arbeit wiederverwendet werden, befindet sich in Kapitel 4. Anschließend ist die Konkrete Umsetzung des neuen Typinferenzalgorithmus in Kapitel 5 beschrieben. Im letzten Abschnitt werden die erreichten Ergebnisse und das geplante weitere vorgehen vorgestellt.

2 Grundlagen

2.1 Java-Compiler

Ein Compiler ist ein Programm das in der Lage ist eine Programmiersprache in eine andere Sprache zu übersetzen. Die meisten Compiler wandeln den für Menschen leichter verständlichen Quellcode in eine für Rechner ausführbare Sequenz von Maschinenbefehlen um. Maschinencode ist dabei meist plattformspezifisch, kann also nur von einem bestimmten Betriebssystem und einer bestimmten Prozessorarchitektur ausgeführt werden. Ein Java-Compiler hingegen übersetzt die Programmiersprache Java nicht direkt in Maschinencode, sondern in einen von der Java-Laufzeitumgebung interpretierbaren Bytecode. Die Java-Laufzeitumgebung ist ein Interpreter für Java-Bytecode, welcher für eine vielzahl von Architekturen angeboten wird. Anstatt den Bytecode in eine andere Sprache zu übersetzen interpretiert die virtuelle Maschine ihn zur Laufzeit und führt die einzelnen Anweisungen nacheinander aus. Dank dieser Vorgehensweise ist ein kompiliertes Java-Programm auf allen unterstützten Plattformen ausführbar. [Her11]

2.2 Lambda-Ausdrücke

Ein Lambda Ausdruck ist ein mit Java 8 neu eingeführtes Sprachkonstrukt. Es erlaubt die Erstellung von anonymen Funktionen innerhalb einer Methode oder Klasse. Ein Lambda Ausdruck darf an allen Stellen zum Einsatz kommen an denen ein Ausdruck erlaubt ist. Er kann also z.B. als Parameter einer Funktion übergeben werden, als Rückgabetyp einer Methode dienen oder einer Variable zugewiesen werden.
Ein Lambda Ausdruck besteht aus zwei Teilen. Auf der linken Seite steht eine Parameterliste, welche Parametertyp und Parameternamen der Funktion auflistet. Auf der rechten Seite befindet sich der eigentliche Funktionsrumpf. Der Rückgabetyp muss nicht angegeben werden. [OP10]

Listing 1: Beispielhafte Benutzung eines Lambda Ausdrucks in einer foreach-Schleife

```
1 interface OneArg<A> {
2   void invoke(A arg);
3 }
4
5 <T> void forEach(Collection<T> c, OneArg<T> block){
6   for (Iterator<T> it = c.iterator(); c.hasNext();){
7     block.invoke(it.next());
8   }
9 }
```

2.3 Funktionales Interface

Ein funktionales Interface ist ein Interface mit nur einer einzigen abstrakten Methode. Ein solches Interface wird auch als SAM-Typ - “Single Abstract Method - Type“ bezeichnet. In der Java Standardbibliothek sind bereits einige SAM-Typ Schnittstellen vorhanden, wie zum Beispiel “Runnable“, “Callable“ oder “EventHandler“. Zusätzlich zu seiner einzigen abstrakten Methode darf ein funktionales Interface noch weitere Methoden besitzen, solange diese nicht als abstrakt definiert sind. Diese Ausnahme ist notwendig, da jedes Objekt und Interface in Java per Definition von der Klasse Object abgeleitet ist und dadurch dessen Methoden erbt.
In Verbindung mit einem Typinferenzalgorithmus spielen funktionale Interface eine besondere Rolle, da ein Lambda Ausdruck als Typ nur ein funktionales Interface annehmen kann. [Naf13]

3 Typ eines Lambda Ausdrucks

Von zentraler Bedeutung für die Konzeption eines Typinferenzalgorithmus für Lambda Ausdrücke ist die Kenntnis über deren Typisierung. Ein Lambda Ausdruck ist bei seiner Definition keinem bestimmten Interface zugeordnet. Je nachdem in welchem Kontext er eingesetzt wird ergibt sich sein Typ. Ein Lambda Ausdruck kann also mehrfach eingesetzt werden und dabei die Typen verschiedener Schnittstellen annehmen. Ein Ausdruck ist zum Typ eines Interface kompatibel, wenn folgende Bedingungen erfüllt sind:

1. Es muss sich um ein funktionales Interface handeln
2. Die Anzahl und Datentypen der Parameter des Lambda Ausdrucks müssen mit denen der SAM-Funktion der Schnittstelle übereinstimmen
3. Die Ausdrücke, welche vom Lambda Ausdruck zurückgegeben werden müssen kompatibel mit dem Rückgabetyp der SAM-Funktion sein.

Wird ein Lambda Ausdruck einer Variable zugewiesen kann diese in alle kompatiblen Interfacetypen gecastet werden.

3.1 Typzuweisung durch den Typinferenzalgorithmus

Bei der Spezifikation des Typinferenzalgoritmus [Plüar] werden unendlich viele funktionale Interfaces definiert. Mit ihnen ist es möglich die Typen aller möglichen Lambda Ausdrücke darzustellen. Diese Interface sind folgendermaßen aufgebaut:

```
1 interface FunN<R, T1, T2, ... , TN> {
2         R apply(T1, ... TN);
3 }
```

Jeder Lambda Ausdruck verfügt über einen Rückgabetyp und N Parameter, welche wiederum jeweils einen Datentyp besitzen. Jedes dieser generischen Interface-Definitionen spezifiziert eine abstrakte Methode namens “apply“. Diese Methode besitzt den Rückgabetyp ’R’ und ’N’ Parameter, welche die Datentypen ’T1’-’TN’ besitzen. Durch korrektes setzen der generischen Typen dieser Interfaces kann zu jedem Lambda Ausdruck ein passendes Funktionales Interface dargestellt werden.

Listing 2: Beispiel

```
1 Fun1<String , String> (string par1) -> {return par1+par1;}
```

3.2 Benutzung von Variablen innerhalb eines Lambda Ausdrucks

Innerhalb eines Lambda Ausdrucks können lokale Variablen deklariert werden. Für deren Gültigkeitsbereich gelten dabei die üblichen Regeln, wodurch diese nur innerhalb der durch den Lambda Ausdruck definierten Funktion gelten. Zusätzlich kann ein Lambda Ausdruck auf die Instanz der Klasse, in welcher dieser deklariert wurde, sowie deren Felder und Methoden zugreifen. Selbst der Zugriff auf lokale Variablen ist unter der Einschränkung, dass diese als "final"deklariert wurden, möglich. Hinweis: Durch eine Neuerung in Java 8 gelten lokale Variablen auch ohne explizite "finalDefinition als "final", sofern diese nur bei ihrer initialisierung beschrieben werden.
Daraus folgt, dass auch Instruktionen innerhalb des Lambda-Ausdrucks für den Typinferenzalgorithmus von Bedeutung sind. Nicht nur zur Inferierung der lokalen Variablen und Parameter der Lambda-Funktion. Auch auf die Typen der Felder und Variablen der umschließenden Klasse kann geschlussfolgert werden, sofern diese im Lambda Ausdruck Verwendung finden.

4 Wiederverwendung des alten Typinferenzalgorithmus

Die Implementierung eines Typinferenzalgorithmus in den Compiler fand bereits 2006 während der Studienarbeit [Bäu05] statt. Eine neue Spezifikation von Martin Plümicke [Plüar], welche im Zuge dieser Studienarbeit umgesetzt wird, ersetzt den alten Algorithmus. Der neue Typinferenzalgorithmus kann aber einige Datentypen der alten Implementierung wiederverwenden. Im folgenden werden die übernommenen Klassen und Methoden aus [Bäu05] vorgestellt:

Pair Die Klasse Pair verknüpft zwei Objektreferenzen miteinander. Sie wurde in der Projektarbeit [Bäu05] zur Speicherung von Typabbildungen eingesetzt. Diese Klasse findet auch im neuen Typinferenzalgorithmus Verwendung. Sie kann eingesetzt werden um Zusammenhänge zweier Objekte darzustellen, was zum Beispiel bei Typabbildungen oder Constraints der Fall ist.

TypeAssumption Die Klasse TypeAssumption stellte im Vorgängerprojekt eine Typannahme dar. Auch der neue Algorithmus arbeitet mit Mengen von Typannahmen, wofür er diese Datenstruktur des Vorgängerprojekts benutzt.

CSet<E> Während der Studienarbeit [Bäu05] wurde bei der Umsetzung des Typinferenzalgorithmus auch eine Datenstruktur zur Speicherung, Vereinigung und Abbildung von Mengen implementiert. Dies war notwendig, da die Java Standardbibliothek keine entsprechende Datenstruktur anbietet. Die Klasse "CSet<E>ïst generisch, kann deshalb jeden beliebigen Datentyp enthalten und bietet sich besonders für eine Wiederverwendung an. Diese Klasse kommt in der Implementierung des neuen Typinferenzalgorithmus unter anderem bei der Speicherung von Typannahmen und Typconstraints (siehe Kapitel 5.2) zum Einsatz.

TypePlaceholder Die Klasse “TypePlaceholder“ ist eine Unterklasse von “Type“ und kann daher im Syntaxbaum als Platzhalter an die Stelle eines Typs gesetzt werden. Nach dem Parsen eines Java-Quellcodes durch den Compiler bleiben die Typangaben zu Elementen im Syntaxbaum leer, zu denen zu diesem Zeitpunkt noch keine Typen bekannt sind. An diese Stellen platziert der Typinferenzalgorithmus TypePlaceholder. Jede Instanz eines TypePlaceholders speichert eine Referenz auf die Variablen und Ausdrücke an denen sie als Typ platziert wurde. Der TypePlaceholder dient also als Observer an dem sich mehrere Subjekte registrieren. Dies erleichtert das Einsetzen der mittels des Typinferenzalgorithmus errechneten Typen an die Stellen der Platzhalter. Hat der Algorithmus den Typ eines Platzhalters ermittelt, kann durch aufrufen einer Methode der entsprechenden TypePlaceholder-Instanz dieser Typ an die am Placeholder registrierten Ausdrücke, Variablen und Anweisungen übermittelt werden. Diese ersetzen den TypePlaceholder durch den berechneten Typ. Dadurch kann ein Typ mit nur einem einzigen Aufruf an alle betroffenen Stellen im Syntaxbaum eingesetzt werden.

5 Typinferenzalgorithmus

TI(Ass, Class(τ, extends(τ'), *fdecls*, *mdecls*)) $=$
let
(Class(τ, extends(τ'), *fdecls*$_t$, *mdecls*$_t$), $ConS$) $=$
TYPE(Ass, Class(τ, extends(τ'), *fdecls*, *mdecls*))
$(\{ (cs_1, \sigma_1), \ldots, (cs_n, \sigma_n) \}, chk)$ $=$ **UNIFY**($ConS$)
in
$\{ (cs_i, \sigma_i($ Class(τ, extends(τ'), *fdecls*$_t$, *mdecls*$_t$) $))\ 1 \leq i \leq n \}$

Abbildung 1: Typinferenzalgorithmus

5.1 Ablauf des Typinferenzalgorithmus

Im folgenden ist der Ablauf des Typinferenzalgorithmus (Abb. 1) beschrieben.

1. Der JavaParser generiert die Klasse “SourceFile“. Sie speichert alle eingelesenen Klassen als “Class“-Objekte, welche jeweils einen abstrakten Syntaxbaum einer Java-Klasse darstellen mit dem der Typinferenzalgorithmus **TI** aufgerufen wird.

2. Die TYPE-Methode erstellt einerseits die benötigten Typ-Annahmen (TypeAssumptions). Dabei handelt es sich um die Felder *fdecls*$_t$ und Methoden *mdecls*$_t$ der geparsten Klassen. Dabei spielt es keine Rolle ob deren Typen bereits bekannt sind. Fehlende Typangaben werden mit TypePlaceholdern ersetzt. Sie werden in einer für den Typinferenzalgorithmus verfügbare Menge von Typannahmen gespeichert. Anschließend ruft er die TYPE-Methode, welche sich in der selben Klasse befindet, auf.

3. Die TYPE-Methode berechnet andererseits die Constraints ($ConS$) der Felder und Methoden des in Schritt 1 erstellten Syntaxbaumes.

4. Ein Unify-Algorithmus [Plü09] unifiziert die gesammelten Constraints. Dabei entsteht eine Lösungsmenge $\{ (cs_1, \sigma_1), \ldots, (cs_n, \sigma_n) \}$ bestehend aus einer Restmenge von

Constraints cs_i, an die keine Bedingung geknüpft ist und einer substitution σ_i als Lösung der unbekannten Typen.

5.2 Berechnen der Constraints

Den Hauptteil des Typinferenzalgorithmus bildet die Berechnung der Constraints. Ein Constraint ist eine Regel, die zwei Typen (darunter fallen auch TypePlaceholder) miteinander in Relation setzt. Der im Zuge dieser Arbeit implementierte Typinferenzalgorithmus erstellt Constraints der Form: "Typ A $<.$ Typ B". Sie legen dabei die Einschränkung fest, dass "Typ A" von "Typ B" erben muss.
Die Berechnung der Constraints beginnt mit dem Aufruf der TYPE-Methode in der Class-Klasse (siehe Kapitel 5.1). Von dieser Methode aus durchläuft eine Tiefensuche den bereits generierten Syntaxbaum des zu inferierenden Java-Quellcodes. Im Endeffekt werden alle Anweisungen und die darin befindlichen Ausdrücke der Klasse nacheinander abgearbeitet. Auf ihnen ruft der Algorithmus die Methoden TYPEStmt (für Anweisungen) oder TYPEExpr (für Ausdrücke) auf. Die Methoden haben Zugriff auf das bisher erstellte ConstraintsSet und die bisher vorhandenen Assumptions. Sie fügen auf Basis der vorhandenen Assumptions neue Constraints hinzu. Hat TYPE den gesamten Syntaxbaum durchlaufen befinden sich alle errechneten Constraints in einem ConstraintsSet.

5.3 Lösen der Constraints

Nach dem Abschluss der Berechnung der Constraints können aufgrund dieser Bedingungen die fehlenden Typen inferiert werden. Diesen Arbeitsschritt übernimmt der Unify-Algorithmus. Dieser wurde bereits in der Studienarbeit [Ott04] implementiert. Die Ausgabe des Unify-Algorithmus ist eine Menge von Pairs (siehe Kapitel 4). Im Erfolgsfall besteht diese Menge aus Paaren, welche jeweils einem TypPlaceholder einen konkreten Typen zuweisen.

5.4 Verarbeiten der Ergebnismenge

Nach einem erfolgreichen Durchlauf des Typinferenzalgorithmus muss im Anschluss die vorliegende Lösungsmenge in den Syntaxbaum übertragen werden. Dazu kann die in der Klasse "TypePlaceholder" enthaltene Funktionalität genutzt werden, welche die mit den Platzhaltern belegten Stellen im Syntaxbaum durch deren errechnete Typen ersetzt. Diese bietet allerdings den entscheidenden Nachteil, dass dieser Vorgang nicht umkehrbar ist. Ein Platzhalter kann somit nur einmalig mit einem Typ getauscht werden. Der Typinferenzalgorithmus ist allerdings so konzipiert, dass sein Ergebnis aus mehreren Lösungen bestehen kann. Das bedeutet, für einen Platzhalter kommen mehrere Typen in Frage. Mit der erwähnten Vorgehensweise könnte nun nur eine einzige Lösung übernommen werden. Zur Kontrolle des Algorithmus sollen jedoch sämtliche Lösungen ausgegeben werden. Dazu wurde eine neue Methode "printJavaCode" eingeführt. Sie durchläuft den Syntaxbaum und generiert dabei den zu der von diesem Baum repräsentierten Klasse zugehörigen Java-Quellcode. Entscheidend ist hierbei, dass der Methode eine der errechneten Lösungen übergeben wird. Die Methode gibt dann aufgrund dieser Informationen den entsprechenden Typ anstatt eines Platzhalters als Java-Quellcode aus. Diese Ausgabe lässt sich für beliebig viele Lösungen wiederholen.

6 Tests

Zur Verifikation der Funktionsweise des Typinferenzalgorithmus werden automatisierte JUnit-Tests durchgeführt. Jeder Test prüft einen bestimmten Teil der implementierten Funktionalitäten anhand eines Testfalls. Die Klasse “mycompiler.mytest.LambdaTest“ bildet dabei die Basis der Testlogik und wird von den einzelnen Testfällen benutzt. Diese Vorgehensweise verhindert Redundanzen im Quellcode und erhöht die Wiederverwendbarkeit. Die einzelnen Testfälle bestehen immer aus zwei Dateien. Einer Datei mit dem zu testenden Java-Quellcode und einem JUnit-Testfall. Zum Testen werden sämtliche JUnit-Testfälle nacheinander aufgerufen. Diese übergeben der “LambdaTest“-Klasse eine Referenz auf den jeweiligen Java-Quellcode und einer HashMap mit den zu erreichenden Ergebnissen. Die “LambdaTest“-Klasse führt den Parser zum Einlesen des Quellcodes aus und stößt anschließend den Typinferenzalgorithmus an. Die übergebene HashMap enthält dabei die vorgegebene Zuordnung von Typen zu Variablen. Der Testfall gleicht abschließend diese Informationen mit den ermittelten Typen des Typinferenzalgorithmus ab und erkennt dadurch Abweichungen.

7 Schlussbetrachtung

7.1 Fazit

Alle in der Spezifikation “Complete Typeinference for Java 8“ (siehe [Plüar]) definierten Algorithmen, außer die den Overloading-Algorithmus (siehe Kapitel 7.2) betreffenden, wurden im Zuge dieser Arbeit umgesetzt. Zusätzlich sind Testfälle und eine Ausgabe zur Anzeige der Inferierten Typen implementiert worden.

7.2 Verzicht auf Overloading

Nicht die gesamte Spezifikation des “Complete Typinference for Java 8“ wurde umgesetzt. So ist die Overloading-Funktion nicht in der Implementierung des Algorithmus enthalten. Diese Funktion berechnet mögliche Überladungen einer Methode und wird zum Ermitteln der Constraints (siehe Kapitel 5.2) eines Methodenaufrufs verwendet. Beim Aufruf einer Methode auf einer Instanzvariablen, deren Typ unbekannt ist, ergeben sich mehrere Möglichkeiten für den Typ der Variablen. Bekannt ist in diesem Moment nur der Name und die Anzahl der Parameter der Methode. Daraus lässt sich eine Auswahl von möglichen Methoden und die damit verbundenen Einschränkungen für den Typ der Instanzvariablen treffen. Da allerdings die Overloading-Logik nicht vollständig implementiert wurde, berücksichtigt der Typinferenzalgorithmus hierbei nur die erste passende Methode, die er findet. Durch diese Einschränkung kann der Typinferenzalgorithmus nicht alle Möglichkeiten zur Typauflösung berechnen.

7.3 Aussicht

Momentan ist die Anwendung in der Lage nur eine einzelne Klasse zu verarbeiten. In Zukunft könnte es möglich sein, ein ganzes Projekt zu verarbeiten und dessen nicht angegebene Typen zu inferieren. Dies hätte den Vorteil, dass zur Ermittlung der Typen nicht nur die Constraints einer einzelnen Klasse herangezogen würden und somit ein besseres Ergebnis entsteht. Zudem ist weder der Overloading-Algorithmus noch ein grafische Benutzeroberfläche entstanden, welche eine praktische Verwendung der entstandenen Anwendung ermöglicht

hätte. Hier könnte zukünftig ein Eclipse-Plugin die in dieser Arbeit entstandene Implementierung des Typinferenzalgorithmus nutzen und eine Benutzerschnittstelle bereitstellen. Dieses könnte ausgelassene Typangaben im Java-Quellcode automatisch ersetzen oder dem Programmierer Typvorschläge offerieren.

Literatur

[Bäu05] BÄUERLE, Jörg: *Typinferenz in Java*. 2005

[Her11] HERDEN, Olaf: *Vorlesung Compilerbau*. 04/2011

[Hol06] HOLZHERR, Timo: *Typinferenz in Java*. 2006

[Lüd07] LÜDTKE, Arne: *Java Typinferenz mit Wildcards*. 2007

[Naf13] NAFTALIN, Maurice: *Lambda FAQ*. `http://www.lambdafaq.org/`, 2013. – [Online; letzter Zugriff 30.01.2013]

[OP10] OPENJDK-PROJECT: *BWorld Robot Control Software*. `http://openjdk.java.net`, 2010. – [Online; letzter Zugriff 30.01.2013]

[Ott04] OTT, Thomas: *Typinferenz in Java*. 2004

[Plü09] PLÜMICKE, Martin: Java type unification with wildcards. In: SEIPEL, Dietmar (Hrsg.) ; HANUS, Michael (Hrsg.) ; WOLF, Armin (Hrsg.): *17th International Conference, INAP 2007, and 21st Workshop on Logic Programming, WLP 2007, Würzburg, Germany, October 4-6, 2007, Revised Selected Papers* Bd. 5437, Springer-Verlag Heidelberg, 2009 (Lecture Notes in Artificial Intelligence), S. 223–240

[Plü10] PLÜMICKE, Martin: *Das JCC-Projket*. `http://www.ba-horb.de/ pl/JCC.html`, 2010. – [Online; letzter Zugriff 30.01.2013]

[Plüar] PLÜMICKE, Martin: *Complete type inference in Java*. to appear

Automated verification of a relation-algebraic matching program

Insa Stucke

Institut für Informatik, Christian-Albrechts-Universität zu Kiel
Christian-Albrechts-Platz 4, D-24118 Kiel
ist@informatik.uni-kiel.de

Zusammenfassung

In this work, we develop a relation-algebraic program for computing a maximum matching in a bipartite graph. The program is based on the algorithm which is a direct consequence of Berge's Lemma. Due to the algebraic nature of the algorithm we prove its correctness with an automated theorem prover.

Generierung von Informationssystemen mit Integritätsbedingungen und Sicherheitseigenschaften

Mathias Weber

Technische Universität Kaiserslautern

1 Einführung

Informationssysteme dienen dazu, elektronische Daten über eine computergestützte Oberfläche dem Benutzer des Systems zur Verfügung zu stellen. Dabei spiegelt sich die Struktur der Daten an vielen Stellen im Informationssytem wieder. Sei es bei der strukturierten Speicherung der Daten, der Definition der entsprechenden Zugriffsrechte oder der Darstellung der Daten auf dem Bildschirm, das Schema der Informationen hat einen maßgeblichen Einfluss auf das Informationssystem.

Dabei sind bei der Erstellung solcher Systeme viele Details zu beachten. In den meisten Fällen gibt es Integritätsbedingungen, die von den Daten nach jeder Operation eingehalten werden müssen. Diese Bedingungen können relativ einfache Einschränkungen des Wertebereichs, allerdings auch komplexere Bedingungen sein, die von einem großen Bereich der Daten abhängen, wie zum Beispiel Summen und Anzahlen von Werten.

Die Anbindung der Daten an die Anwendung muss die Integrität des Datenbestandes nach jeder modifizierenden Aktion sicherstellen. Die Überprüfungen selbst zu schreiben und aufrechtzuerhalten ist relativ komplex und fehleranfällig. Fehler an dieser Stelle können zu erheblichem Verlust von Informationen durch Korrumpierung des gesamten Datenbestandes führen. Die Wiederherstellung der Daten ist ein zeitaufwändiger Prozess.

Eine weitere Eigenschaft, die durch ein Informationssystem garantiert werden muss, ist die Vertraulichkeit der Daten. Unbefugte Zugriff müssen unter allen Umständen vermieden werden. Gleichzeitig muss die Rechtezuordnung jedoch einfach genug sein, um den Überblick zu behalten und dynamisch genug sein, um einfaches Rechtemanagement zu ermöglichen. Um die Rechte zur Laufzeit anpassen zu können, müssen die Informationen über die Benutzer ebenfalls Teil des Datenbestandes sein. Möchte man nun die Zugriffsrechte Implementieren, stößt man auf ähnlich Probleme wie schon bei der Prüfung der Integrität der Daten. Die Umsetzung ist komplex, da die Überprüfungen an allen Stellen im System gemacht werden müssen, an denen auf den Datenbestand zugegriffen wird. Gleichzeitig können Fehler bei der Umsetzung zu erheblichen Schäden führen, da sensible Daten nach außen gegeben werden können.

Der von uns gewählte Ansatz verwendet eine Beschreibung der Struktur und der zusätzlichen Bedingungen, die für die Daten gelten müssen und generiert daraus Datenhaltungs- und Sicherheitsschicht.

2 Aktueller Stand

Es existiert bereits ein Prototyp, der eine abgewandelte Form eines Relax NG [3] Schemas entgegennimmt, welches mit Integritätsbedingungen der Daten annotiert ist. Der Generator erstellt daraus ein Databinding, welches es ermöglicht, dem Schema entsprechende Daten aus einer XML-Datei zu lesen und auf diese aus Java zuzugreifen. Außerdem können in der Schemabeschreibung Prozeduren angegeben werden, welche Modifikationen auf den existierenden Daten durchführen können. Da diese Prozeduren die Integritätsbedingungen erhalten müssen, wird vor der Ausführung der Prozedur geprüft, ob das Ergebnis der Operationen wiederum die Integrität der Daten erhält [2]. Die Vorbedingungen der Prozeduren werden aus den Operationen, die im Rumpf der Prozedur ausgeführt werden, generiert. Der Generierungsprozess der Vorbedingungen ist in Isabelle/HOL spezifiziert und komplett verifiziert [1, 2].

Der so generierte Datenkern erlaubt lesende Zugriff auf alle Werte der Datenhaltung. Modifikationen der Daten können jedoch nur durch Aufruf einer im Schema spezifizierten Prozedur erfolgen. Damit wird sichergestellt, dass die Integrität der Daten durch alle Operationen erhalten werden, die durch den Programmierer aufgerufen werden können. Eine Verletzung dieser Bedingungen ist somit nicht möglich. Ein großer Vorteil des verwendeten Ansatzes ist es, nicht das gesamte Schema in die Prüfung mit einzubeziehen. Statt dessen wird eine schwächste Vorbedingung berechnet, die vor der Ausführung der Prozedur gelten muss, um die Einhalten der Datenintegrität zu gewährleisten.

3 Ausblick und verwandte Arbeiten

Für die Realisierung der Sicherheitseigenschaften gibt es bereits Frameworks, die eine Vereinfachung versprechen. Allerdings sind diese häufig Teil eines größeren Frameworks, was bei Benutzung erhebliche Anpassungen und Einschränkungen der Software nach sich zieht. Beispiele solcher Frameworks sind Spring Security [4] und EJB [5].

Diese Ansätze erlauben in einer bestimmten Granularität die benötigten Rechte zu definieren, die für die Benutzung von bestimmten Teilen des Systems benötigt werden. Dabei ist man bei EJB jedoch darauf eingeschränkt, die Software mit Hilfe von EJB Beans zu realisieren. Die so erzeugte Software setzt jedoch einen EJB Container voraus, was eine große Einschränkung der Einsatzgebiete der Software nach sich zieht. Eine Entkopplung der Software von EJB ist nicht möglich, ohne ein anderes Framework zur Realisierung der Sicherheitseigenschaften zu benutzen.

Das Framework von Spring Security spezialisiert sich auf die Realisierung von Web Anwendungen und versteht sich als Ersatz von Enterprise Java Technologien. Mit Hilfe dieser Technologie ist es recht einfach, Web Anwendungen auf Basis ihrer URLs zu sichern. Eine Loginseite zur Authentifizierung wird von Spring Security automatisch erstellt. Allerdings reicht diese Sicherung nicht aus, da die Implementierung der Seite selbst und damit der Zugriff auf die Informationen nicht überprüft wird. Eine Alternative ist die Definition von Methoden-Aufrufen, die überprüft werden sollen. Dabei müssen die Objekte, deren Methodenaufrufe überprüft werden sollen jedoch durch das Spring-Framework verwaltet sein, was nur durch Nutzung des Dependency Injection [6, 7] Teils von Spring erreicht werden kann. Die gesamte Umsetzung der Software muss somit an diese Art der Architektur angepasst werden, was nicht immer erwünscht oder möglich ist. Außerdem bekommt man bei dieser Umsetzung keine Hilfe dabei, welche Methodenaufrufe geprüft werden sollen und welche nicht für die Sicherheit der Anwendung relevant sind.

Meine Forschung zielt darauf ab, die Schicht, die die Sicherheit der Zugriffe auf den Datenkern gewährleisten soll, aus der Spezifikation des Schemas zu generieren. Dies ist grundsätzlich möglich, da der Datenkern selbst aus dem gleichen Schema generiert wird und damit bekannt ist, welche Methoden welchen Teilbäumen der Daten entsprechen. Die Spezifikation der Sicherheitseigenschaften wie vorhandene Benutzer und Rollen bauen auf den Daten, die durch den Datenkern verwaltet werden, auf und ist somit dynamisch zur Laufzeit anpassbar. Die Definition der Zugriffsrechte soll als weitere Annotation des Schemas erfolgen, was eine intuitive Zuordnung zwischen Zugriffsrechten und geschützten Daten ermöglicht.

Literatur

1. P. Michel, A. Poetzsch-Heffter: Verifying and Generating WP Transformers for Procedures on Complex Data, ITP 2012, August 13-15, 2012, Princeton, New Jersey, USA.
2. P. Michel, A. Poetzsch-Heffter: Maintaining XML Data Integrity in Programs - An Abstract Datatype Approach, SOFSEM 2010, January 23-29, 2010, Špindleruv Mlýn, Czech Republic.
3. http://relaxng.org/; Letzter Zugriff am 28.08.2013.
4. Spring Security. http://static.springsource.org/spring-security/site/index.html.
5. Enterprise JavaBeans Technology. http://www.oracle.com/technetwork/java/javaee/ejb/index.html.
6. JSR 330: Dependency Injection for Java. http://jcp.org/en/jsr/detail?id=330.
7. M. Fowler: Inversion of Control Containers and the Dependency Injection Pattern. http://www.martinfowler.com/articles/injection.html

Typprüfung mittels Attributgrammatiken auf natürlichsprachlichen Anforderungs-Beschreibungen technischer Systeme

Sebastian Wendt, Wolf Zimmermann

4. September 2013

Martin-Luther-Universität Halle
sebastian.wendt@informatik.uni-halle.de
wofl.zimmermann@informatik.uni-halle.de

Zusammenfassung

Entwicklung von Soft- und Hardwareprodukten benötigt solides Requirements-Engineering. Diese Arbeit behandelt das Extrahieren von Anforderungen aus Fließtext in natürlicher Sprache. Dazu setzen wir Schablonen ein, die natürliche Sprache in eine Form mit gerade soviel Formalismus zwingt, dass Anforderungen sowohl maschinenlesbar als auch für den innerbetrieblichen Schriftverkehr geeignet ist, und der Autor außerdem gezwungen ist mehrdeutige Formulierungen (z.B. subjektlose Passivsätze) aufzulösen.

Wir verwenden ein Typsystem, um die Kombination von Verben und Substantiven nachvollziehbaren und eindeutigen Regeln zu unterwerfen, und ein einheitliches Verständnis des Modells bei allen Stakeholdern zu erreichen.

Keywords: Attributgrammatik, Boilerplates, Schablonen, NLP

1 Einleitung

Requirements-Engineering ist ein notwendiger Bestandteil der Produktentwicklung von Soft- und Hardwareprodukten. Im industriellen Umfeld mit vielen Stakeholdern und mehrjährigen Entwicklungszyklen ermöglicht es RE, rückblickend die Gründe für eine getroffene Entscheidung zu ermitteln – sog. Traceability bzw. Verfolgbarkeit – und vorausschauend Aufgaben zu strukturieren und zu verteilen.

Für RE können eine Vielzahl von Modellierungsmethoden, meist grafischer Form, eingesetzt werden. Aber auch werden viele Anforderungen als Fließtext geschrieben, um sie in Prozessdokumenten wie Pflichtenheften (Bugtracker-Einträgen, Beratungsprotokollen, usw.) zu verwenden. Fließtext wird vor allem von jenen bevorzugt, die keine Kenntnisse im Umgang mit Modellierungsmethoden besitzen oder die mit solchen Personen kommunizieren müssen.

Die vorliegende Arbeit ist ein Baustein einer Forschungskooperation mit einem Industriepartner. Unser Ziel ist es, kausale Abhängigkeiten zu extrahieren und mittels ihrer den Zusammenhang von textuellen Anforderungen untereinander und bis hin zu physischen Produkten zu verfolgen. Da probabilistische Parser auf unbeschränkter natürlicher Sprache nur mit eingeschränkter Genauigkeit arbeiten[1], benutzen wir Schablonen, um den Aufwand für die automatische Textverarbeitung soweit zu senken, dass kontextfreie Grammatiken ausreichen die relevanten Satzbestandteile eindeutig zu identifizieren. Gleichzeitig bleibt der Text lesbar, so dass er ohne Änderungen z.B. in Pflichtenheften verwendet werden kann.

Anschließend überprüfen wir mittels eines Typsystems und einer Attributierten Grammatik die Einhaltung von Konventionen bei der Kombination der identifizierten Satzbestandteile gemäß eines domänenspezifischen Glossars. Die Arbeit bildet die Grundlage für weitere Arbeiten zur Analyse kausaler Beziehungen zwischen Anforderungen und Produkten.

[1] z.B. F1-Wert von 90.4% für den Stanford Parser auf dem WSJ-Korpus laut [7]

Unsere Arbeit an Requirements-Engineering ist motiviert durch ein Forschungsprojekt mit einem Industriepartner, der uns Beispieldokumente aus einem aktuellen Projekt bereitgestellt hat. Ziel ist es, eine Datenflussanalysearchitektur zum Zwecke der Herstellung von Verfolgbarkeit auf den Projektdaten zu etablieren. Fragestellungen, die Verfolgbarkeit mittels Datenflussanalyse beantworten soll, sind:

- Sind die Anforderungen vollständig?
- Sind die Anforderungen widerspruchsfrei?
- Wurden die spezifizierten Anforderungen vollständig umgesetzt?
- Warum wurden bestimmte Lösungen auf bestimmte Anforderungen so gewählt, wie sie gegenwärtig vorzufinden sind?

Diese Arbeit präsentiert eine Lösung für die ersten beiden Fragen im Kontext eines einzelnen Satzes. Die Fragen, die wir tatsächlich beantworten werden lauten:

- *Vollständigkeit:* Sind die Verben im Anforderungstext mit mindestens der richtigen Anzahl an Argumenten (Substantiven) versehen?
- *Konsistenz:* Sind die Verben im Anforderungstext mit Argumenten eines yu den Beschriftungen passenden Typs versehen?

Die Bedeutung von Argumenten, Beschriftungen und Typen erklären wir in Abschnitt 3.2. Die Beantwortung aller vier Fragen im Kontext des Gesamtprodukts verfolgen wir in späteren Arbeiten.

2 Verwandte Arbeiten

Die vorliegende Arbeit beschreibt die syntaktische und die semantische Analyse der Fließtexte. Probabilistische Parser aus der natürlichen Sprachverarbeitung, maschinelles Lernen und Methoden des Information Retrieval nähern sich der Informationsextraktion aus Anforderungen mit probabilistischen Herangehensweisen.

Hull et.al. [4] stellen Schablonen für die Beschreibung von Anforderungen in natürlicher Sprache unter dem Namen *Boilerplates* vor. Das Grundmuster von Boilerplates ist: „The <system> shall be able to <function>“, wobei system und function Platzhalter sind. Das Grundmuster wird erweitert durch Einschränkungen („... not less than ...“, „... of type ... within ...“).

Mavin et.al. [5] verwenden Boilerplates und zeigen wie Anforderungsbeschreibungen mit ihrer verbessert werden können. Sie benötigen lediglich 5 Schablonen. Die Verbesserung messen sie anhand der Merkmale „Untestbarkeit“, „Vorauseilende Implementierung“ (Anforderungen, die eine Implementierung vorgeben), „Geschwätzigkeit“, „Duplikation“, „Auslassung“, „Komplexität“, „Vagheit“ und „Mehrdeutigkeit“[2]. Sowohl ihr Transformations- als auch ihr Messverfahren muss jedoch manuell durchgeführt werden.

Cleland et.al. [3] benutzen *Goal-Centric Traceability*, um damit ein städtisches System zur Einsatzkoordinierung von Räumfahrzeugen zu verbessern. Sie benutzen GCT als Methode des Information-Retrieval, um es den Entwicklern des Systems zu ermöglichen, Abhängigkeiten von nichtfunktionalen Anforderungen zu anderen Anforderungen herauszufinden (z.B. „Das System soll eine Anfrage innerhalb von 10 Sekunden beantworten.“ beeinflusst „Das System soll Daten komprimiert speichern.“) und das System entsprechend zu verbessern. Da das Verfahren probabilistisch arbeitet, um Korrelationen zu erkennen, bedarf es ständiger Korrektur durch die Benutzer und ist nicht in der Lage selbständig von Korrelationen auf kausale Abhänigkeiten zu schließen.

Arora et.al. [2] stellen ein Werkzeug zur Verarbeitung von Boilerplates namens „Rubric“ vor. Die eingesetzten Schablonen stammen von Rupp und liegen als BNF-Grammatik vor. Eine Java-Implementierung ist öffentlich verfügbar[3]. Der Schwerpunkt liegt auf der Erkennung der Konformität von Texten mit den Schablonen und setzt diese nicht via einer Kontextfreien Grammatik voraus. Daher ist das eingesetzte Textanalyseverfahren eine Kombination aus heuristischen und statistischen Methoden. Ihr Verfahren erreicht einen F1-Wert von 90.9%.

Ambriola et.al. [1] entwickelten einen Domänenbasierten Parser namens „Cico“ für das Anforderungs-Analysesystem „Circe“. Cico arbeitet im Gegensatz zum Parser in der vorliegenden Arbeit heuristisch, d.h. er akzeptiert

[2]Untestability, Inappropriate Implementation, Wordiness, Duplication, Omission, Complexity, Vagueness, Ambiguity

[3]https://sites.google.com/site/rubricnlp/

auch „unbehandelte“ Fließtexte, wobei die Genauigkeit abhängig ist von der Verträglichkeit des jeweiligen Textes mit den Beispielen im Trainingskorpus des Parsers.

Gildea et.al. [6] geben einen Überblick über Namensschemata aus der natürlichen Sprachverarbeitung für die in Abschnitt (3.2) verwendeten Namen für die Labels/Beschriftungen (*agent*, *receiver*, usw.). Da unsere Arbeit davon abstrahiert, ist die Wahl des Schemas für das sogenannte *Semantic Role Labeling* für unsere Zwecke beliebig.

Im Vergleich zu probabilistischen Herangehensweisen, mit dem Zweck Korrelationen zu entdecken und zu modellieren, ist es unser Ziel, eindeutige kausale Abhängigkeiten zu erfassen und zu verarbeiten. Als Voraussetzung dafür erachten wir erachten die Anwendung strikter Schablonen als die geeignete Grundlage für die Erfassung und Typisierung von Satzbestandteilen in natürlicher Sprache.

3 Parsen von Fließtext

Parsen von Text in unbeschränkter natürlicher Sprache ist ein offenes Problem. Da wir aber an einer domänenspezifischen Lösung mit dem Industriepartner arbeiten, können wir Vorgaben machen, um das Problem zu beschränken. Wir stellen zuerst eine kontextfreie konkrete Grammatik für Schablonen vor und verarbeiten sie anschließend mittels einer Attributgrammatik, um Vollständigkeit und Konsistenz auf Satzebene zu zeigen.

3.1 Schablonengrammatik

Mittels Schablonen wollen wir viele syntaktische Schwierigkeiten natürlicher Sprache ausschließen, wie sie in Anforderungstexten auftreten, wo die Tendenz zur Hyperkorrektheit besteht. Diese äußert sich z.B. durch den Einsatz inverser, subjektloser Konjunktivsätze (*Tritt Überlastung auf, schaltet der Generator ab.*) oder stichpunktartige Schreibweise (*Parameter X benötigt, mit Nachkommastelle.*). Syntaktisch schwer zu behandelnde Konstrukte wie separierbare Verben (*A stellt B dar*) und Pronomen (*Damit kann oben Genanntes unterschieden werden.*) lassen sich damit auch vermeiden. Wir beschränken einen Satz weitgehend auf eine Menge von Nominalphrasen und Verben, die ggf. noch durch Konjunktionen (*Wenn-dann*) eingegrenzt werden. Nominalphrasen folgen dem pseudo-regulären Muster[4]:

```
<Artikel> <^Nomen>* <Nomen>+
```

und lassen sich deshalb mit einer kontextfreien Grammatik problemlos abbilden.

Die konkrete Grammatik ist im Folgenden abgebildet und besteht im Kern aus Statements, die entweder bedingte oder unbedingte Soll-Anforderungen beschreiben:

```
decl : cond_stmt .
decl : uncond_stmt .
```

Die unbedingten Anforderungen müssen das Wort *soll* enthalten, so dass sie leicht im Text erkennbar sind:

```
uncond_stmt : e_agent 'soll' e_verb t_whether e_action '.' .
uncond_stmt : e_agent 'soll' e_action '.' .
uncond_stmt : e_agent 'soll' 'gleichzeitig' e_action '.' .

t_whether : o_comma 'dass' | o_comma 'ob' .
o_comma : ',' | .
```

Unterstützung für Kann-Anforderungen ist leicht nachzurüsten. Bedingte Anforderungen müssen die Wörter *Wenn* und *dann* im Text enthalten:

```
cond_stmt : t_if e_action t_then 'soll' uncond_stmt_i .
uncond_stmt_i : e_agent e_action '.' .
t_if : 'Wenn' | 'Falls' .
t_then : o_comma 'dann' | o_comma 'so' .
```

Konditionalsätze mit führendem Verb entfallen. Aus: *Kräht der Hahn früh auf dem Mist, ändert sich das Wetter oder es bleibt wie es ist.* wird: *Wenn der Hahn auf dem Mist früh kräht, dann soll das Wetter sich ändern oder das Wetter soll wie es ist bleiben.*

Der handlungsausführende Agent (`e_agent`, z.B. „Der Benutzer *soll … betätigen“, „Das Programm* soll … melden“) muss in jedem Satz explizit durch eine Nominalphrase (`np`) definiert werden:

[4] <^Nomen> meint „nicht-Substantive“. Der Ausdruck entspricht etwa: `/(der|die|das) ([^A-Z][a-z]+)* ([A-Z][a-z]+)+/`

```
e_agent : np .
np : det mods nseq | 'sich' | 'sich selbst' .
mods : word mods | .
nseq : capword | capword nseq .
```

Die übrigen Verbargumente sind ungeordnet vor dem Verb aufgelistet:

```
e_action : nps e_verb .
e_verb : word .
nps : np | np nps .
```

`det` (Artikel), `word` (kleinbuchstabige Wörter) und `capword` (Wörter, die mit Großbuchstaben beginnen oder in Anführungszeichen stehen) sind Terminalsymbole aus der lexikalischen Analyse.

3.2 Typsystem

Haben wir einen Satz dieser Grammatik geparst, wollen wir nun mit den Informationen aus dem Syntaxbaum überprüfen, ob Verben und Nominalphrasen „zusammenpassen", d.h. ob die Zuordnung vollständig (alle Argumente sind vorhanden) und konsistent (Argumente haben den richtigen Typ) ist. Wir wollen damit erreichen, dass nicht nur die Syntax von Anforderungen, sondern auch die domänenspezifische Semantik, also der Gebrauch und die Kombination von bedeutungstragenden Elementen (Substantiven und Verben) nachvollziehbaren und nicht mehrdeutigen Regeln folgt.

Wir definieren ein einfaches Typsystem, bestehend aus Wertetypen T und Beschriftungs-Typen V.

- Menge der Typen Θ: ein Typ $T \in \Theta$ wird identifiziert mit einem Substantiv, das eine Kategorie beschreibt, z.B. $Benutzer$ oder $Funktion$. Ein Typ ist zugleich eine Menge der Eigennamen und Bezeicher die Instanzen (Werte) dieser Kategorie sind, z.B. *Herr Müller*[5] und *Drehzahlbegrenzung*. Die Menge aller Typen ist eine Teilmenge der Potenzmenge aller Instanzen: $\Theta \subset \mathcal{P}(\{\textit{Herr Müller}, \textit{Drehzahlbegrenzung}, \ldots\})$. Da die Menge der Substantive und Eigennamen in einer Sprache unbeschränkt ist, ist es auch die Menge der Typen
- Menge der Verben Υ: ein Verb entspricht einer geordneten Liste von notwendigen Beschriftungen $V \in \Upsilon$. Die Elemente von V sind Tupel aus Beschriftungen $r \in R$ und a-posteriori Typ $T \in \Theta$, z.B. $(agent, Person)$. Die Grundgesamtheit der Beschriftungen R ist beliebig aber fest. Gängige Beschriftungs-Schemata wie in [6] verwenden ca. 10 Beschriftungen[6], von denen nur jeweils 2–4 an einem Verb zu finden sind. Für das Beispiel in dieser Arbeit sei $R = \{agent, receiver, manipulated\}$. Wir fordern, dass Beschriftungs-Typen nicht Teil des Typsystems für Substantive sind, also $\underset{T \in \Theta}{\forall} T \cap R = \emptyset$.

Obwohl die Reihenfolge der Argumente in natürlicher Sprache nicht fest ist, definieren wir auf der Menge R eine beliebige **totale Ordnung**, und fordern, dass Argumente im Satz in Reihenfolge verwendet werden. In unserem Beispiel gelte: $agent < receiver < manipulated$. Diese Ordnung gibt sowohl die Reihenfolge der Verwendung im Text an als auch die Reihenfolge der Elemente in V, wovon wir bei der Auswertung in Abschnitt 3.4 Gebrauch machen. Wir fordern außerdem, dass jedes Argument bzw. sein Typ eindeutig einer Beschriftung zugeordnet werden kann, und dass allen Beschriftungen $R_V = \{r : (r, t) \in V\}$ ein Typ zugeordnet wird. So ist z.B. der folgende Satz zulässig:

$$\underbrace{\text{Das Programm}}_{agent} \text{ soll } \underbrace{\text{dem Benutzer}}_{receiver} \underbrace{\text{eine Nachricht}}_{manipulated} \text{ anzeigen.} \qquad (1)$$

Der nächste Satz ist jedoch nicht zulässig, da kein typkompatibles Argument für die Position `manipulated` exisitert:

$$\text{*}\underbrace{\text{Das Programm}}_{agent} \text{ soll } \underbrace{\text{dem Benutzer}}_{receiver} \underbrace{\text{einen Benutzer}}_{?} \text{ anzeigen.} \qquad (2)$$

3.3 Typhierarchie

Damit eine Verbdefinition nicht alle möglichen Typen explizit aufführen muss, benötigen wir ein Typsystem, mit dem wir Substantive in Äquivalenzklassen zusammenfassen können. Da unser Typsystem auf Wertemengen basiert, fügen wir Obertypen einfach als Vereinigung der gewünschten Elemente bzw. Untermengen zu Θ hinzu. Um eine Untertypsbeziehung in unserer Sprache abzubilden reicht eine einfache Syntax aus:

[5] Rollennamen, wie *Maschinist* sind Typen mit Eigennamen als Elementen

[6] vorgeschlagen werden *Agent*, *Patient*, *Theme*, *Experiencer*, *Beneficiary*, *Instrument*, *Location*, *Source* und *Goal*. Das Schema von FrameNet/SALSA kennt 624 Beschriftungen, darunter aber viele idiosynkratische wie `White_house`

```
Eine Fehlermeldung ist eine Nachricht .
```

Die nötigen Regeln fügen wir unserer Schablonengrammatik aus Abschnitt 3.1 hinzu:

```
decl: isa_stmt .
isa_stmt : np 'ist' np '.' .
```

Damit ist beschrieben, dass der Typ $Fehlermeldung$ an den Typ $Nachricht$ anpassbar ist, also $Fehlermeldung \subset Nachricht$. Der folgende Satz ist ohne Änderung der Definition für *anzeigen* gültig:

$$\underbrace{\text{Das Programm}}_{agent} \text{ soll dem } \underbrace{\text{Benutzer}}_{receiver} \text{ eine } \underbrace{\text{Fehlermeldung}}_{manipulated} \text{ anzeigen.}$$

Ein minimales Typsystem für dieses Beispiel enthält die Werte:

$$\begin{aligned}
\Theta =& \{Programm, Benutzer, Fehlermeldung, Nachricht, \bot, \top\} \\
Programm =& \{\text{Programm}\} \\
Benutzer =& \{\text{Benutzer}\} \\
Fehlermeldung =& \{\text{Fehlermeldung}\} \\
Nachricht =& \{\text{Nachricht, Fehlermeldung}\} \\
\bot =& \{\} \\
\top =& \bigcup_{T \in \Theta} T \\
V =& \{U_{anzeigen}\} \\
U_{anzeigen} =& \{(agent, Programm), (receiver, Benutzer), (manipulated, Nachricht)\}
\end{aligned}$$

Die Typhierarchie sollte ein Baum, muss mindestens aber ein gerichteter azyklischer Graph sein. Da die Typhierarchie mengenbasiert ist, lässt sich leicht feststellen, wenn Zyklen eingeführt werden: zwei Typen $T, T' \in \Theta$ sind dann identisch. Diese Fehlerquelle beim Arbeiten mit mehreren Autoren, hervorgerufen durch Missverständnisse und ein unzureichendes Glossar (z.B. A definiert: *Eine Funktionalität ist eine Kundenfunktion.* B definiert: *Eine Kundenfunktion ist eine Funktionalität.*) kann also erkannt und durch den Übersetzer gemeldet werden.

3.4 Auswertung

Da wir Literale (Substantive) als Typen betrachten, und Typen als Wertemengen, ist es mittels Mengenoperationen einfach möglich, Obertypen zu bestimmen. Die Operationen $\sqcup$ und $\sqsubseteq$ sind dabei identisch mit den Mengenoperatoren $\cup$ bzw. $\subseteq$ über Θ. Wollen wir eine (entsprechend der Ordnung der Beschriftungen R) geordnete Liste von Argumenttypen $T_A = \langle Programm, Benutzer, Fehlermeldung \rangle$ auf Zuweisbarkeit an das Verb V untersuchen, lassen sich die relevanten Eigenschaften leicht überprüfen. Die Indizes i markieren dabei die Position in den Listen T_A und V.

$$\begin{aligned}
\text{Vollständigkeit: } & |T_A| = |V| \\
\text{Konsistenz: } & \forall T_i : (r_i, T_i) \in V \land t_i \in T_A \land t_i \sqsubseteq T_i
\end{aligned}$$

Vollständigkeit bedeutet, die exakte Zahl notwendiger Argumente vorhanden ist. Konsistenz bedeutet, dass jeder Typ an i-ter Position in der Argumentliste ein Untertyp des Typs T_i ist, der in der Verbdefinition an i-ter Stelle mit der Beschriftung r_i assoziiert ist. Mehrdeutigkeit bei der Zuweisung von Argumenten kann nicht auftreten, da die Ordnung der Beschriftungen und Argumente fest vorgegeben ist.

3.5 Vergleich mit Ontologien

Linguistische Ontologien werden in der natürlichen Sprachverarbeitung benutzt, um Daten über den Sprachgebrauch zu sammeln und z.B. für das Training probabilistischer Parser zu verwenden. Eine Ontologie kann ein Graph sein, bestehend aus Ecken, die Substantiven entsprechen, und Kanten, die Verben entsprechen. Eine Kante (bzw. Hyperkante falls sie mehr als 2 Ecken hat) in der Ontologie gibt an, dass die Verwendung des Verbs mit den Argumenten an ihren Enden zulässig ist. Dasselbe haben wir in der Verbdefinition in Abschnitt 3.2 beschrieben. Im

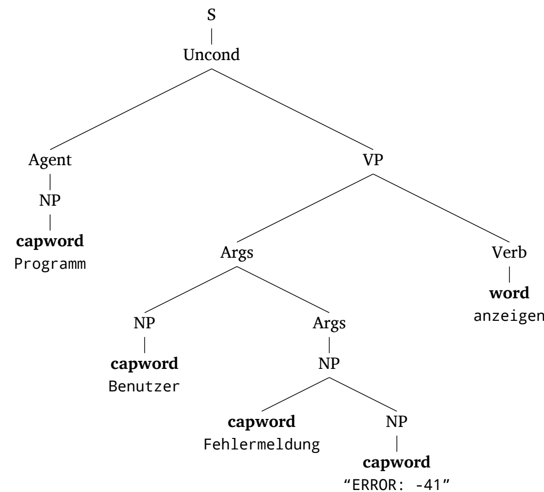


Abbildung 1: Abstrakter Syntaxbaum für Typprüfungs-Beispiel in Satz 3

Unterschied zu Ontologien sparen wir durch die Typhierarchie den Aufwand, für jedes Tupel von Argumenten einen Eintrag in die Ontologie vornehmen zu müssen, und modellieren äquivalente Argument in Klassen über die `is_a`-Beziehung. Außerdem ist unsere Verb- und Typdefinition per Konvention vollständig: nicht modellierte Verben oder Substantive verursachen einen Fehler, im Gegensatz zu Ontologien, die wegen ihren probabilistischen Verwendung keinen Anspruch auf Vollständigkeit erheben.

4 Auswertung mit Attributierter Grammatik

Wir führen das Beispiel fort und demonstrieren jetzt die Typprüfung mittels einer Attributierten Grammatik. Basierend auf der Schablonengrammatik aus Abschnitt 3.1 legen wir folgende abstrakte Syntax zugrunde:

```
S ::= Uncond .
S ::= Cond .
Uncond ::= Agent VP .
Cond ::= VP Uncond .
Agent ::= NP .
VP ::= Args Verb .
NP :: = capword | capword NP .
Args ::= NP | NP Args .
Verb ::= word .
```

Die Nichtterminale der konkreten Grammatik auf der rechten Seite werden auf die abstrakten Nichtterminale auf der linken Seite abgebildet. Von Terminalen wird weitestgehend abstrahiert.

```
MAPSYM
S ::= decl .
Uncond ::= uncond_stmt .
Cond ::= cond_stmt .
Agent ::= e_agent .
NP ::= np .
VP ::= e_action .
Args ::= nps .
Verb ::= e_verb .
```

Wir betrachten den Beispielsatz:

$$\underbrace{\text{Das Programm}}_{agent} \text{ soll } \underbrace{\text{dem Benutzer}}_{receiver} \text{ die } \underbrace{\text{Fehlermeldung “ERROR: -41”}}_{manipulated} \text{ anzeigen.} \qquad (3)$$

Abbildung 1 zeigt den abstrakten Syntaxbaum für den Beispielsatz. Die Nominalphrase *die Fehlermeldung* “ERROR: -41” enthält zwei Substantive, von denen eines als Typ definiert ist. Mittels $type$ erhalten wir die in der Definitionstabelle verzeichneten Typen: $Fehlermeldung$ und $\perp$ (da nicht definiert). Der a-priori Typ der Nominalphrase, zu speichern im Attribut pri, ergibt sich aus der paarweisen Bildung des Supremums ($\sqcup$) der Untertypen.

S
$types = \langle (agent, Programm), (receiver, Benutzer), (manipulated, Fehlermeldung) \rangle$
Uncond
$types = \langle (agent, Programm), (receiver, Benutzer), (manipulated, Fehlermeldung) \rangle$
Agent
$pri = Programm$
NP
$pri = Programm$
capword
`Programm`
VP
$env = key($`anzeigen`$)$
$pris = \langle Benutzer, Fehlermeldung \rangle$
Args
$pris = \langle Benutzer, Fehlermeldung \rangle$
Verb
$env = key($`anzeigen`$)$
word
`anzeigen`
NP
$pri = Benutzer$
capword
`Benutzer`
Args
$pris = \langle Fehlermeldung \rangle$
NP
$pri = Fehlermeldung \sqcup \bot$
capword
`Fehlermeldung`
NP
$pri = \bot$
capword
`"ERROR: -41"`

Abbildung 2: Typprüfungs-Beispiel mit Attributberechnungen für Satz 3

Aus Platzgründen beschränken wir uns im Beispiel auf eine Nominalphrase mit Eigennamen. In der Praxis ist es sinnvoll, ggf. sogar zu erzwingen, dass jedem Typ ein Eigenname oder Bezeichner folgt. Denn soll die Satz-Analyse nicht alleine, sondern im Kontext einer Anforderungsbeschreibung erfolgen, dann benötigen wir diese Eigennamen um satz- und anforderungsübergreifende Zusammenhänge herzustellen.

Die a-priori Typen aller Nominalphrasen sind in `Args` als Liste im synthetisierten Attribut $pris$ enthalten. Da die Reihenfolge der Argumente im Syntaxbaum vorgeschrieben ist, können wir in `Uncond` annehmen, dass ein Typ in $pris$ entweder der Reihenfolge der erwarteten Beschriftungen (*getLabels*) entspricht oder fehlerhaft ist. Da sich die erwarteten Beschriftungen je nach Verb unterscheiden können und in der Ordnung nicht aufeinanderfolgen müssen, verpaaren wir (*zip*) zwecks Identifikation die erwarteten Beschriftungen mit den Typen, und legen das Ergebnis in $types$ ab.

Die Vollständigkeits- und Konsistenzprüfung findet ebenfalls in `Uncond` statt: Vollständigkeit ist gegeben, wenn die Anzahl der Typen bzw. Nominalphrasen (*length*) mit der erwarteten Anzahl der Beschriftungen (*getLabels*, abzüglich des Agenten) übereinstimmt. Konsistenz ist gegeben, wenn für alle Beschriftungen des Typsystems (nicht nur die am aktuellen Verb vorhandenen) gilt, dass ihr a-priori Typ (*priType*) an den a-posteriori Typ (*postType*) aus der Verbdefinition anpassbar ist ($\sqsubseteq$). Wird eine Beschriftung durch das Verb nicht gefordert, so ist der a-posteriori Typ $\bot$, was bedeutet, dass nur nicht belegte Argumente ($\bot$) auf ihn anpassbar sind. Beispiel 4 illustriert diesen Fall.

Der abstrakte Syntaxbaum für Beispiel 3 mit allen berechneten Attributen ist in Abbildung 2 zu sehen. Die Berechnungsvorschriften zur Attributberechnung im Beispiel 3 lauten im Einzelnen:

---

rule S ::= Uncond
attr S.*types* ← Uncond.*types*
rule Uncond ::= Agent VP
attr Uncond.*types* ← zip(getLabels(VP.env), cons(Agent.*pri*, VP.pris))
cond length(VP.*pris*) = length(getLabels(VP.*env*)) − 1
priType(Uncond.*types*, label('agent')) ⊑ postType(VP.*env*, label('agent'))
priType(Uncond.*types*, label('receiver')) ⊑ postType(VP.*env*, label('receiver'))
priType(Uncond.*types*, label('manipulated')) ⊑ postType(VP.*env*, label('manipulated'))
rule VP ::= Args Verb
attr VP.*pris* ← Args.*pris*
VP.*env* ← Verb.*env*

rule	Args ::= NP Args
attr	Args_1.*pris* ← cons(NP.*pri*, Args_2.*pris*)
rule	Args ::= NP
attr	Args.*pris* ← cons(NP.*pri*, nil)
rule	Agent ::= NP
attr	Agent.*pri* ← NP.*pri*
rule	NP ::= **capword** NP
attr	NP_1.*pri* ← type(**capword**) ⊔ NP_2.*pri*
rule	NP ::= **capword**
attr	NP.*pri* ← type(**capword**)
rule	Verb ::= **word**
attr	Verb.*env* ← key(**word**)

Die verwendeten Attribute haben die Typen:

```
syn pri: Type   syn pris: ⟨Type⟩   syn types: ⟨Label×Type⟩   syn env: DefTableKey
```

Wir benutzen die folgenden Hilfsfunktionen:

`Type type( ID )`
holt den Typ für ein Substantiv oder ⊥ wenn nicht vorhanden

`DefTableKey key( ID )`
holt die Unterschlüssel für ein Verb

`Beschriftung label( String )`
konvertiert eine Zeichenkette in eine Beschriftung

`Type priType( Liste[Beschriftung×Type] types, Beschriftung label )`
holt den a-priori Typ für eine Verb-Beschriftung aus `types`, oder ⊥ wenn nicht enthalten

`Type postType( DefTableKey verb, Beschriftung label )`
holt den a-posteriori Typ für eine Verb-Beschriftung aus der Definitionstabelle, oder ⊥ wenn nicht für das Verb definiert

`List[Beschriftung] getLabels( DefTableKey verb )`
holt die Liste notwendiger Beschriftungen für ein Verb

`bool ⊑( Type a, Type b )`
testet auf Anpassbarkeit von `a` nach `b`

`Type ⊔( Type a, Type b )`
findet das Supremum von `a` und `b`

`Liste[T] cons( T hd, Liste[T] tl )`
Listenkonstruktur zusammen mit `nil`

`Liste[T×U] zip( Liste[T] t, Liste[U] u )`
fügt zwei Listen elementweise zu einer Liste von Tupeln zusammen

`int length( Liste[T] l )`
bestimmt die Länge der Liste `l`

Wir betrachten den folgenden Beispielsatz, der die Beschriftung *receiver* nicht verwendet:

$$\underset{agent}{\text{Der Benutzer}} \text{ soll } \underset{manipulated}{\text{das Programm}} \text{ starten.} \tag{4}$$

Der abstrakte Syntaxbaum mit den berechneten Attributen für Beispiel 4 ist in Abbildung 3.3 dargestellt. Das Typsystem ist dasselbe wie in Abschnitt 3.2. Hinzugekommen ist die Verbdefinition:

$$U_{starten} = \{(agent, Programm), (receiver, \bot), (manipulated, Programm)\}$$

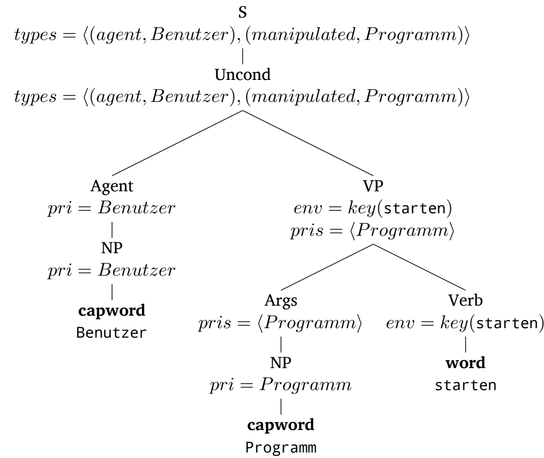


Abbildung 3: Typprüfungs-Beispiel mit Attributberechnungen für Satz 4

Die Attributberechnungen im Knoten `Uncond` für Satz 4 sehen wiefolgt aus:

$$\begin{aligned}
\text{Uncond.}\textit{types} &\leftarrow \text{zip(getLabels(VP.}\textit{env}\text{), cons(Agent.}\textit{pri}\text{, VP.}\textit{pris}\text{))}\\
&\leftarrow \text{zip(} \langle agent, manipulated\rangle, \langle Benutzer, Programm\rangle \text{)}\\
&\leftarrow \langle (agent, Benutzer), (manipulated, Program)\rangle\\
\text{length(VP.}\textit{pris}\text{)} &= \text{length(getLabels(VP.}\textit{env}\text{)) - 1}\\
\text{length(} \langle Programm\rangle \text{)} &= \text{length(} \langle agent, manipulated\rangle \text{) - 1}\\
1 &= 2 - 1\\
\text{priType(Uncond.}\textit{types}\text{, label('agent'))} &\sqsubseteq \text{postType(VP.env, label('agent'))}\\
Benutzer &\sqsubseteq Benutzer\\
\text{priType(Uncond.}\textit{types}\text{, label('receiver'))} &\sqsubseteq \text{postType(VP.env, label('receiver'))}\\
\bot &\sqsubseteq \bot\\
\text{priType(Uncond.}\textit{types}\text{, label('manipulated'))} &\sqsubseteq \text{postType(VP.env, label('manipulated'))}\\
Programm &\sqsubseteq Programm
\end{aligned}$$

5 Zusammenfassung und Ausblick

Wir präsentierten in dieser Arbeit einen bekannten Ansatz zur Aufbereitung von Anforderungen, nämlich mittels Schablonen. Wir vereinfachten natürliche Sprache mittels Schablonen so, dass die Satzstruktur mit einer kontextfreien Grammatik erkannt werden kann und der Text gleichzeitig für Pflichtenhefte u.dgl. geeignet bleibt.

Dann präsentierten wir eine neue Interpretation von Ontologien als Typsystem. Durch das Typsystem und Berechnungen in einer attributierten Grammatik sind wir in der Lage, für einzelne Verben und dazugehörige Substantive bzw. Nominalphrasen zu überprüfen, ob ihre Semantik den Kriterien der Vollständigkeit und Konsistenz genügt, also Beschriftungs- und Argumenttypen zusammenpassen. Wir schufen dazu eine Beschreibungssprache für Verb-Signaturen sowie für Äquivalenzrelationen von Substantiven.

Wir sind dadurch in der Lage, die domänenspezifische Semantik eines Glossars zu modellieren und Texte mittels eines Parser auf Einhaltung des Modells zu überprüfen. Gegenüber Ontologien zeichnet sich unser Ansatz darin aus, dass unsere Aussagen zur Modellkorrektheit nicht statistisch sondern normativ per expliziter Definition begründet sind, und dass die Modellierung mittels Typhierarchien einfacher, schneller und zuverlässiger als ist, als das Zusammensammeln von Korrleationen aus annotierten Korpora.

Ausblick

Der aktuelle Ansatz eignet sich zur Analyse der domänenspezifischen Textdokumente bei unserem Industriepartner, da uns diese ohne Anforderungs- und Implementierungs-Kontext vorlagen, also noch nicht im Sinne durchgehenden Requirement-Engineerings ohne Modellbruch – von der Kundenanforderung bis zum ausgelieferten Produkt – miteinander in Beziehung gesetzt sind. In weiterführenden Arbeiten haben wir deshalb vor, erstens Anforderungs-Dokumente und Code in Beziehung zu setzen, um die Umsetzung von Anforderungen zu überprüfen, und zweitens rückwärts von Pflichtenheft-Anforderungen auf Kundenanforderungen in Textform, zu schließen, um so die verbleibenden Fragen aus unserer Zielstellung in Abschnitt 1 zu beantworten. Dazu wird es notwendig sein, nicht nur die semantische Kohärenz von Textdokumenten in sich selbst (gemäß der Typ- und Verbdefinitionen), sondern auch die Kohärenz der Typen mit übergeordneten, abstrakteren Anforderungen bzw. konkreten Typen einer Programmiersprache zu betrachten.

Danksagung

Die Arbeit findet im Rahmen des Forschungsvorhabens ELSY: „Effizientes Design Elektronischer Systeme“, gefördert vom Bundesministerium für Wissenschaft und Forschung (16M3202D), statt. Dank gilt ebenso dem Projektträger VDI/VDE-IT.

Literatur

[1] V. Ambriola and V. Gervasi. Processing natural language requirements. In *Proceedings of the 12th international conference on Automated software engineering (formerly: KBSE)*, ASE ’97, pages 36–, Washington, DC, USA, 1997. IEEE Computer Society.

[2] Chetan Arora, Mehrdad Sabetzadeh, Lionel Briand, Frank Zimmer, and Raul Gnaga. Rubric: a flexible tool for automated checking of conformance to requirement boilerplates. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 599–602, New York, NY, USA, 2013. ACM.

[3] Jane Cleland-Huang, Raffaella Settimi, Oussama BenKhadra, Eugenia Berezhanskaya, and Selvia Christina. Goal-centric traceability for managing non-functional requirements. In *Proceedings of the 27th international conference on Software engineering*, ICSE ’05, pages 362–371, New York, NY, USA, 2005. ACM.

[4] Elizabeth Hull, Ken Jackson, and Jeremy Dick. *Requirements Engineering*. Springer, 2005.

[5] Alistair Mavin and Philip Wilkinson. BIG EARS - The return of “Easy Approach to Requirements Syntax”. *18th IEEE International Requirements Engineering Conference*, 2010.

[6] Martha Palmer, Daniel Gildea, and Nianwen Xue. *Semantic Role Labeling*. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers, 2010.

[7] Richard Socher, John Bauer, Christopher D. Manning, and Andrew Y. Ng. Parsing with compositional vector grammars. In *Proceedings of ACL*, 2013.

Syntaxanalyse auf Wiedervorlage
(Erweiterte Zusammenfassung)

Baltasar Trancón y Widemann[1 2]
Markus Lepper[2]

[1] Technische Universität Ilmenau
[2] <semantics /> GmbH Berlin

1 Einführung

Die automatisierte Implementierung von Parsern auf Basis von formalen Grammatiken ist seit langem theoretisch sehr gründlich erforscht und praktisch in zahlreichen ausgereiften Werkzeugen realisiert. Allerdings fällt auf, dass die praktisch eingesetzten Grammatik-Notationen im Vergleich zu anderen domänenspezifischen Sprachen auf relativ niedrigem Abstraktionsniveau arbeiten. Wir machen Vorschläge zur Verbesserung der Situation, und diskutieren diese an ersten Erfahrungen mit dem Prototypen unseres Parser-Generatorwerkzeugs branch, das klassische Analyse- und Implementierungstechniken mit Innovationen kombiniert, die modernen Anforderungen der Softwaretechnik gerecht werden sollen.

2 Entwurf

Das Werkzeug branch ist prinzipiell der Familie der $LL(k)$-Parser-Generatoren zuzuordnen.

- Echte $LL(k)$-Analyse wird mit geeigneten Datenstrukturen und bekannten Vereinfachungstechniken aus dem Compilerbau hinreichend effizient implementiert, um den Generator wahlweise als statisches oder dynamisches Werkzeug zu betreiben.
- Die erzeugten top-down Parser kommunizieren ihre Ergebnisse über eine strukturiert ereignis-basierte Schnittstelle (vergleichbar XML SAX[3]), die neben traditioneller Erzeugung von Ableitungsbäumen auch inkrementelle Verarbeitung von Eingaben mit geringer Speicherkomplexität und Latenz erlaubt.
- Die traditionelle Schwachstelle der top-down Parsierung, Linksrekursion, wird durch eine neuartige Technik behandelt, die allerdings vermutlich äquivalent zu in anderen Kontexten bekannten Lösungen ist.
- Zentrales Forschungsthema ist die Maximierung des Abstraktionsniveaus und damit des Gültigkeitsbereichs und der Wiederverwendbarkeit von Syntaxregeln. Dabei gehen wir über den Stand der Kunst (modulare Grammatiken wie in SDF[2] oder Vererbung von Regeln wie in ANTLR[4]) hinaus, und untersuchen das Potential von Synaxregeln höherer Ordnung, die individuell unter anderem mit anderen Syntaxregeln parametrisiert werden können.

Der letztere Aspekt soll an einem Beispiel erläutert werden.

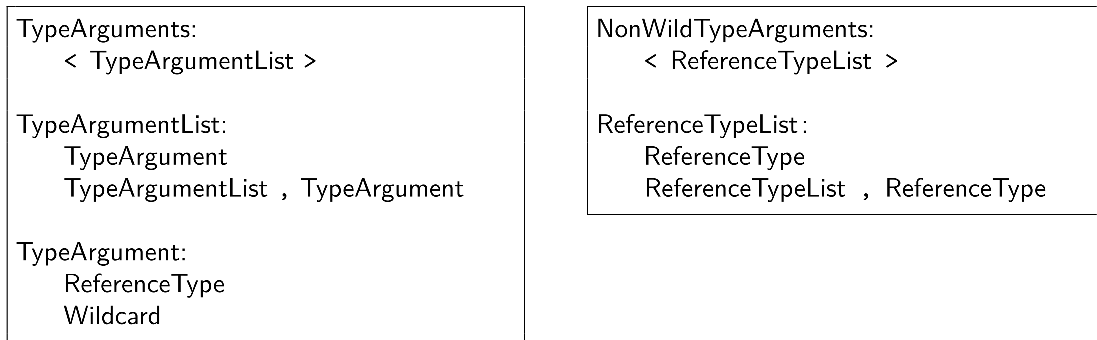


Abbildung 1: Ausschnitte aus der Java-Spezifikation [1], Abschnitte 4.5.1 bzw. 8.8.7.1.

3 Beispiel

Dem grundlegenden Formalismus von Syntax-Berschreibungen, kontextfreien Grammatiken, fehlen aus pragmatischer Sicht Abstraktionsmittel: Analogien zwischen unterschiedlichen syntaktischen Kategorien einer komplexen Sprache, oder Gemeinsamkeiten von Sprachfamilien, können nicht effektiv ausgedrückt werden. Existierende Ansätze zur Modularisierung und Vererbung von Grammatiken haben dieses Problem nicht gelöst. Auch beim Einsatz von Abstraktionen der Implementierungs-Plattform von Parser-Kombinatorbibliotheken herrscht in der Praxis große Zurückhaltung.

Eine vielversprechende Gelegenheit zur Abstraktion, die bisher kaum Beachtung gefunden hat, ist die Parametrisierung einzelner Regeln mit einfachen Daten (als Ausdruck kombinatorischer Variationen) oder anderen Regeln (als Ausdruck struktureller Analogien). Die Operatoren von EBNF und verwandten Formaten können als fest vorgegebener Vorrat von rudimentären parametrischen Regeln angesehen werden. Können solche aber vom Benutzer definiert und benannt werden, ergeben sich ganz neue Ausdrucksmöglichkeiten.

Zur Illustration und Motivation dieser Technik, die ein zentrales Merkmal unseres Prototypen ist, sind in Abbildung 1 zwei inhaltlich zusammenhängende aber getrennt behandelte Fragmente der Spezifikation von Java abgebildet. Dabei beschreibt das zweite Fragment eine Teilsprache des ersten (nähmlich ohne Instanzen von Wildcard).

Verschiedene Umstände erschweren es, diese beiden Fragmente als hochgradig verwandte Instanzen eines gemeinsamen Musters zu erkennen. Zum einen sind die beiden Fragmente räumlich weit voneinander entfernt. Zum anderen enthalten sie Idiosynkrasien auf verschiedenen Ebenen:

- Listen über einer syntaktischen Kategorie beinhalten nicht die begrenzenden Klammern, der Plural der entsprechenden Kategorie dagegen schon. Dieselbe Nomenklatur gilt für aktuelle Parameter eines Methodenaufrufs (Abschnitt 15.12), nicht aber für formale Parameter einer Methodendeklaration (Abschnitt 8.4.1).
- Listen und ihre Elemente sind im linken Beispiel nach ihrer *semantischen Rolle* (TypeArgument), im rechten dagegen nach ihrem *syntaktischen Inhalt* (ReferenceType) benannt.

Allgemeinere Regeln, die sowohl dem Designer einer Sprache bei zukünftigen Revisionen als auch dem Leser einer Sprachspezifikation beim Erlernen der Syntax gute Dienste leisten könnten, sind in dem hier verwendeten Formalismus kontextfreier Regeln erster Ordnung nicht ausdrückbar. Dabei existieren solche Regeln für die Sprache Java durchaus; die hier betroffenen sind:

```
rule List [open, sep, close, elem, empty : bool] = open (elem (sep elem)∗ | empty⇒) close

rule JavaList [open, close, elem, empty : bool] = List [open, ",", close, elem, empty]

rule TypeList [elem] = JavaList ["<", ">", elem, false ]

rule TypeArguments[wild : bool] = TypeList[TypeArgument[wild]]
rule TypeArgument [wild : bool] = ReferenceType | wild⇒ Wildcard
```

Abbildung 2: Abstraktion auf verschiedenen Ebenen in branch.

- Elemente einer Liste (beliebiger Kategorie) werden durch Komma getrennt.
- Typ-Parameter-Listen stehen in spitzen Klammern und sind nicht leer.

Solche Regeln aus der Dokumentation der Grammatik abzuleiten, ist typischerweise recht schwer. Wir vermuten darin einen der wesentlichen Gründe, dass Sprachspezifikationen beim Erlernen der Sprache in der Praxis kaum eine Rolle spielen.

Dabei ist den Designern der Sprache Java keine besondere Nachlässigkeit zu unterstellen. Kontextfreie Grammatik als Ausdrucktmittel ist zwar theoretisch ausreichend, aber pragmatisch unangemessen primitiv für die Beschreibung einer so komplexen Sprache wie Java. Analog zum Einsatz höherer Programmiersprachen für die Implementierung komplexerer Algorithmen, wäre hier ein höheres Niveau der Spezifikation von Syntax dringend erforderlich.

Abbildung 2 zeigt eine im Hinblick auf allgemeine Aussagekraft refaktorisierte Form des Grammatik-Fragments. Die beiden erwähnten allgemeineren Regeln sind durch die Produktionen JavaList bzw. TypeList abgebildet. Die Produktion List spezifiziert eine von Java unabhängige allgemeinere Regel, die auch in vielen anderen Kontexten werden kann; tatsächlich gibt es in manchen EBNF-Varianten einen entsprechenden Operator. Der Handhabbarkeit von solchen Operatoren sind aber enge Grenzen gesetzt, was Anzahl und Spezifizität angeht. Der Ansatz parametrischer Regeln ist von diesem Problem nicht betroffen.

Eine vollständige Diskussion der verwendeten Notation würde hier zu weit führen. Ein technischer Bericht, der die Notation definiert und in Selbstanwendung illustriert, ist in Arbeit.

- Eine Regel hat die Form **rule** x = y, wobei x ein Nonterminal mit formalen Parametern, und y ein EBNF-artiger Ausdruck ist.
- Als Parameter eines Nonterminals (in eckigen Klammern) kommen Nonterminale (auch höherer Ordnung, also selbst parametrisch; nicht-parametrischer Fall unmarkiert), die primitiven Datentypen **nat** und **bool**, sowie benutzerdefinierte Aufzählungstypen in Frage.
- Ein Guard-Ausdruck der Form c⇒ ist äquivalent zur trivialen Sprache $\{\varepsilon\}$ falls c = **true** gilt, andernfalls zur leeren Sprache. Ebenfalls die leere Sprache bezeichnen Instanziierungen von Nonterminalen mit undefinierten Parameter-Ausdrücken, wie etwa negativen Zahlen.

Bei einer Grammatik mit parametrischen Regeln handelt es sich natürlich nicht mehr um eine kontextfreie Grammatik im eigentlichen Sinn. Da unsere Absicht nicht die Erweiterung der theoretischen Sprachklasse, sondern die Verbesserung der Beschreibungsmittel ist,

betrachten wir die pure kontextfreie Grammatik, die sich durch rekursive Instanziierung ausgehend von einem nicht-parametrischen Startsymbol ergibt. Syntaxbeschreibungen, bei denen diese Rekursion nicht terminiert, die also nicht direkt einer endlichen kontextfreien Grammatik entsprechen, werden als ungültig abgelehnt. Dies äußert sich im aktuellen Prototypen durch Nichttermination der Transformation, soll aber später durch geeignete Terminationsanalysen ergänzt werden.

4 Fazit

Obwohl die Blütezeit der gegenseitigen Befruchtung von Theorie und Praxis der Syntaxanalyse in den 1960er und 70er Jahren stattgefunden zu haben scheint, ist das Fach als Forschungsgebiet noch keineswegs erledigt. Einerseits machen moderne Rechner viele traditionelle Heuristiken und Komplexitäts-Vorurteile obsolet. Andererseits bringen moderne Kontexte der Softwaretechnik (z.B. interaktive Programme, Unicode, Familien domänenspezifischer Sprachen) auch neue Herausforderungen mit sich. Über der technischen Notwendigkeit der Suche nach neuen Implementierungstechniken sollte dabei nicht das Streben nach Eleganz und dem richtigen Abstraktionsniveau aus dem Auge verloren werden. Beim gleichzeitigen Verfolgen beider Ziele, wie in unserem Prototypen `branch`, ergeben sich interessante Synergien und wertvolle Hinweise für zukünftige Forschung.

Literatur

[1] James Gosling u. a. *The Java Language Specification, Java SE 7 Edition*. Addison-Wesley, 2013. ISBN: 978-0133260229.

[2] J. Heering u. a. “The syntax definition formalism SDF—reference manual—”. In: *SIGPLAN Not.* 24.11 (1989), S. 43–75. ISSN: 0362-1340. DOI: 10.1145/71605.71607.

[3] David Megginson. *SAX 2*. 2004. URL: http://www.saxproject.org/.

[4] Terence Parr. *ANTLR v4*. 2012. URL: http://www.antlr.org/.

Information about the Author(s)

Roswitha Picht, Wolf Zimmermann (Hrsg.)
Institut für Informatik
Universität Halle
06099 Halle, Germany
E-Mail: roswitha.picht@informatik.uni-halle.de
E-Mail: wolf.zimmermann@informatik.uni-halle.de
WWW: http://www.informatik.uni-halle.de/mitarbeiter/picht/
WWW: http://www.informatik.uni-halle.de/mitarbeiter/zimmermann/