Combining Tools and Languages for Static Analysis and Optimization of High-Level Abstractions[1]

Markus Schordan

Vienna University of Technology, Austria
markus@complang.tuwien.ac.at

Abstract. We present an approach for combining different analysis and transformation tools that enables their application to popular programming languages without extending existing compilers. Analysis results are made available as annotations of a common high-level intermediate representation and as generated source code annotations. We also support an external file format. The presented Static Analysis Integration Engine allows the selection of an arbitrary tool chain from the pool of integrated tools, most suitable for a certain program analysis or manipulation task. The architecture is evaluated with an implementation targeting full C++, considering templates, object-oriented features, as well as low-level features. The integrated tools are the LLNL-ROSE source-to-source infrastructure, the Program Analyzer Generator from AbsInt, and the language Prolog for manipulating terms representing C/C++ programs.

1 Motivation

For instrumentation tools, source-to-source optimizers, slicing tools, refactoring tools, and tools for enabling code comprehension, it is important to keep the source-code structure available for presenting the results of source code manipulating operations to the user. It is important that the results can be easily put into relation to the original program. This aids the user of such a tool, but complicates the internal handling of the source program during analysis and transformation because the results must be mapped back to the original program. Compilers usually translate the input programs to a lower-level representation for reducing the number of different language constructs, allowing to keep a program analysis more compact. The presented approach aims at utilizing compiler technology

[1] To appear in Post-Workshop Proceedings of the 24th Workshop of GI Fachgruppe Programmiersprachen und Rechenkonzepte, 2007.

but without losing syntactic or semantic information about the original input program. Therefore all tools are integrated to operate on, or map forth and back to a high-level intermediate representation. The goal is to permit building arbitrary tool chains from the pool of integrated tools.
In Section 2 we present the architecture of our Static Analysis Tool Integration Engine (SATIrE) allowing a seamless integration of powerful tools. The concrete implementation is presented in Section 3, also describing each tool and how it is integrated in SATIrE. In Section 4 we discuss related tool-based infrastructures and in in Section 5 we provide a short overview of the perspectives that we anticipate for the extensibility of our approach.

2 Architecture

The architecture of the Static Analysis Tool Integration Engine (SATIrE) is shown in Fig. 1. An essential aspect is that information gathered about an input program can be generated as annotation in the output program, and that the output program can again serve as input program. This allows to make analysis results persistent as generated source-code annotations. Utilizing such annotations, it allows to perform whole program optimization.
The architecture shown in Fig. 1 consists of the following kinds of components

Front End. The input language, L, is translated to a high-level intermediate representation (HL-IR).
Annotation Mapper. The annotations in L are translated to annotations of the HL-IR.
Tool IR Builder. Each tool may require its own IR. The Tool-IR Builder creates the required Tool-IR by translating the HL-IR to the Tool-IR.
Tool. A tool analyzes or transforms its respective Tool-IR.
Tool IR Mapper. The Tool-IR mapper either maps the Tool's IR back to High-Level IR or maps the computed information or results back to locations in the HL-IR.
Program Annotator. The HL-IR annotations are translated to a representation in the source code. This can be comments, pragmas, or some specific language extension.

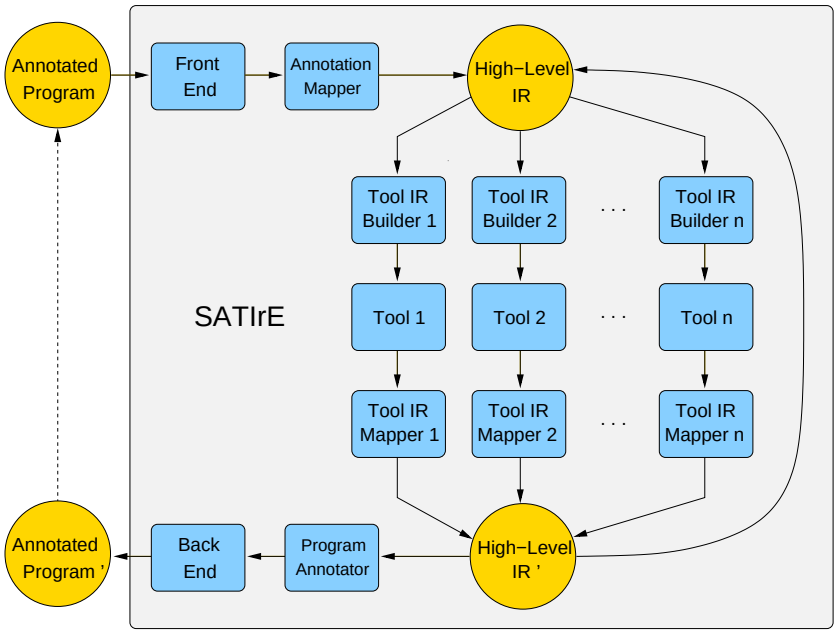

Fig. 1. Static Analysis Tool Integration Engine Architecture

Back End. From the HL-IR a program in language L is generated (including annotations).

To allow a seamless integration of the tools, the Annotation Mapper, Program Annotator, the Tool-IR Builders and Tool-IR Mappers are offered by SATIrE. In Fig. 1 the solid back-edge represents an iterative application of the tools within SATIrE.
For example, library source codes can be analyzed and the library's interface source code can be annotated with analysis results. When the library is used by an application, the library annotations can then be utilized by the application optimizer. We have demonstrated the optimization of the use of a parallel C++ array abstraction and achieved similar performance as with an equivalent Fortran implementation [9].

3 Integrated Tools and Languages

To date we have integrated the Program Analyzer Generator PAG [7], which generates analyzers from high-level specifications,

the LLNL-ROSE infrastructure for source-to-source transformation of C++ programs [12], and a term representation of programs suitable for a Prolog interpreter, into SATIrE. In the following sections we describe each integrated tool and give a short overview of its integrated components.

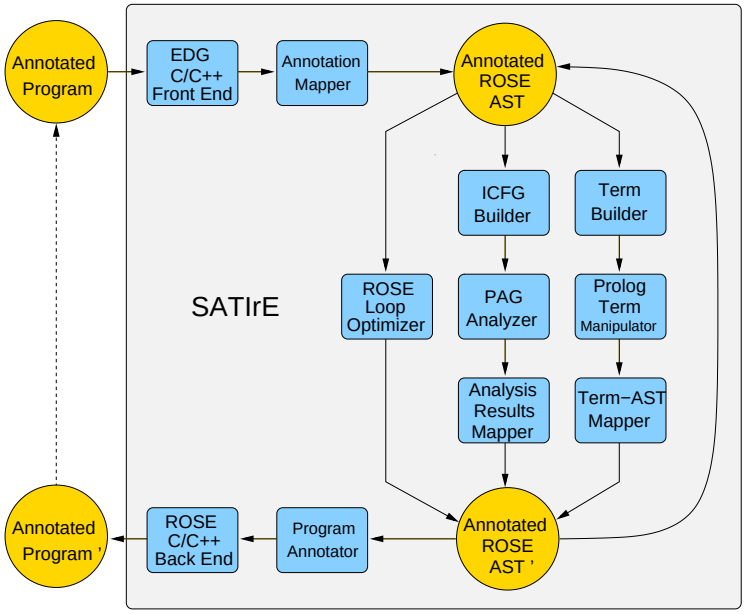

Fig. 2. Static Analysis Tool Integration Engine Implementation

3.1 LLNL-ROSE Integration

The LLNL-ROSE infrastructure offers several components to build a source-to-source translator. The ROSE components integrated into SATIrE are

C/C++ Front End. ROSE uses the Edison Design Group C++ Front End (EDG) [3] to parse C++ programs. The EDG Front End generates an abstract syntax tree (AST) and performs a full type evaluation of the C++ program. The AST is represented as a C data structure. ROSE translates this data structure into a decorated object-oriented AST (ROSE-AST).

Abstract Syntax Tree (ROSE-AST). The ROSE-AST represents the structure of the input program. It holds additional information such as the type information for every expression, exact line and column information, instantiated templates, the class hierarchy (as it can be computed from the input files), an interface that permits querying the AST, an an attribute mechanism for attaching user-defined information to AST nodes.

C/C++ Back End. The Back End unparses the AST and generates C++ source code. It can be specified to unparse all included (header) files or the source file(s) specified on the command line with include-directives. This feature is important when transforming user-defined data types.

Loop Optimizer. The loop optimizer was ported by Qing Yi from the Fortran-D compiler to directly operate on the ROSE-AST. It supports a wide range of loop transformations such as loop fusion, loop fission, loop skewing, loop interchange and blocking that can be applied to a given ROSE-AST.

3.2 Program Analyzer Generator Integration

The Program Analyzer Generator (PAG) from AbsInt, takes as input a specification of a program analysis and generates an analyzer that implements the analysis. The analyzer operates on an interprocedural control flow graph (ICFG) and provides the computed analysis results as C data structure as well as a visualization of the ICFG and the analysis results. The components necessary for a seamless integration of PAG into SATIrE are

ICFG Builder. Creates the inter-procedural control flow graph (ICFG) for a given ROSE-AST.

PAG Analyzer. Generated by the Program Analyzer Generator (PAG) from a user-defined analysis specification using the OPTLA language.

Analysis Results Mapper. Maps the analysis results back to locations in the ROSE-AST and makes them accessible as ROSE-AST annotations.

Various types of ICFG attributes (for example numeric labels for statements) and support functions are provided to the analyzer by

appropriate functions. Thus, the high-level analysis specification can access any information the ROSE-AST provides, such as types of expressions, the class hierarchy, etc.

3.3 Example

A short example output of an automatically annotated program is shown for the post-processed results of a shape analysis [10] in Fig. 3. The shape analysis is specified using PAG, the input program is a C++ program implementing a list reversal (and other list operations). After translating the C++ program to the corresponding ROSE-AST, SATIrE's ICFG builder creates the ICFG. Then the PAG analyzer performs the shape analysis and the Analysis Results Mapper maps the results back to the ROSE-AST. A post-processing of the computed shapes generates may and must alias information. The aliasing results are attached to the ROSE-AST nodes as must/-may alias annotations. The Program Annotator generates from the AST the annotations as source code comments, and the ROSE Back End generates the annotated C++ code.
The actual parameter in the call to the function `reverseList` is `l`, and is therefore aliased with the formal parameter `x`. When post analysis information (after a statement) and pre analysis information (before a statement) is the same, it is shown in the same line and preceded with `post,pre`.

3.4 Prolog Integration

The integration of Prolog allows to specify a manipulation of the AST as term manipulation. The SATIrE components necessary for integration are

Term builder. Creates a term representation for a given AST. The term representation is complete, meaning that it contains all information available in the AST. The term representation is stored in an external file.

Prolog term manipulator. The term manipulation is specified as Prolog rules.

Term-AST Mapper. The transformed term is read in and translated to a ROSE-AST.

```
class List* reverseList(class List* x)
{
  // pre must_aliases : {(l,x)}
  // pre may_aliases : {(l,x)}
  class List* y;
  // pre,post must_aliases : {(l,x)}
  // pre,post may_aliases : {(l,x)}
  class List* t;
  // post,pre must_aliases : {(l,x)}
  // post,pre may_aliases : {(l,x)}
  y = ((0));
  // post must_aliases : {(l,x)}
  // post may_aliases : {(l,x)}
  // pre must_aliases : {}
  // pre may_aliases : {(l,t),(l,x),(l,y),(l,y->next),(t,y->next)}
  while(x != ((0))) {
    // pre must_aliases : {}
    // pre may_aliases : {(l,t),(l,x),(l,y),(l,y->next),(t,y->next)}
    t = y;
    // post,pre must_aliases : {(t,y)}
    // post,pre may_aliases : {(l,t),(l,x),(l,y),(l,y->next),(t,y)}
    y = x;
    // post,pre must_aliases : {(x,y)}
    // post,pre may_aliases : {(l,t),(l,x),(l,y),(x,y)}
    x = (x -> next);
    // post,pre must_aliases : {(x,y -> next)}
    // post,pre may_aliases : {(l,t),(l,y),(x,y->next)}
    y -> next = t;
    // post must_aliases : {}
    // post may_aliases : {(l,t),(l,y),(l,y->next),(t,y->next)}
  }
  // post,pre must_aliases : {}
  // post,pre may_aliases : {(l,t),(l,x),(l,y),(l,y->next),(t,y->next)}
  t = ((0));
  // post,pre must_aliases :  {}
  // post,pre may_aliases : {(l,x),(l,y),(l,y->next)}
  return y;
  // post must_aliases : {}
  // post may_aliases : {(l,x),(l,y),(l,y->next)}
}
```

Fig. 3. Example of a C++ program, annotated automatically with must/may aliasing information which is computed by a post-processing phase from the results of a shape analysis [10]. We extended the shape analysis to an inter-procedural analysis. The analysis is specified by using PAG's specification language.

This approach has been successfully adopted within the COSTA project for performing Worst-Case Execution Time Analysis for a given C program. A detailed description can be found in [8].

4 Related Work

Glynn et al. show that support for program understanding in development and maintenance tasks can be facilitated by program analysis techniques [4]. They outline the addition of generic program analysis support to a generic, language-based software development environment.
Harrold and Rothermel present a technique for separate analysis of modules [5]. The work focuses on one particular analysis, interprocedural may alias analysis, but the design of the analyzer is general and similar to our setting. For inter-procedural analysis an ICFG is created. The separation in control flow and intermediate representation of statements and expressions is the same as in our approach. The analysis is a modular analysis, meaning that a module is a set of interacting procedures or a single procedure that has a single entry point. The approach allows to reuse the analysis results after analyzing a module and thus, is applicable to large scale software and real world applications. In our approach we can add analysis results as annotations to source-code, allowing to reuse analysis results in a subsequent analysis step. This can either be done on the IR-level or the annotated source code is read in again.
For optimizing compilers the automatic generation of data flow analyses and optimizations out of concise specifications has been a trend for several years. The systems of [1, 2] concentrate on "classical" inter-procedural optimizations, whereas the system of [13] is particularly well suited for local transformations based on data dependency information. We integrated PAG because it is a tool that allows to generate analyzers from specifications for similar analysis problems. In our infrastructure the transformation of the program is performed by utilizing the AST rewrite capabilities of ROSE and by using Prolog for term manipulation.
In [6] a technique is presented for automatically proving compiler optimizations *sound*, meaning that their transformations are always semantics-preserving. The domain specific-language Cobalt allows

to specify optimizations to operate on a C-like intermediate representation. The implemented correctness checker interfaces with the automatic theorem prover Simplify. A similar setting could be added to our infrastructure by integrating also tools for checking and automatic proving into our current PAG-ROSE environment. Addressing the additional needs of such tools and leveraging its benefits is a driving force in the development of SATIrE.

5 Conclusions and Perspectives

We have presented SATIrE that allows to combine tools for analysis and transformation. The Front End translates the possibly annotated input program to a high-level representation (HL-IR). This HL-IR is translated to an appropriate Tool-IR for each integrated tool. The results computed by the respective tool are always mapped back to the common HL-IR. The HL-IR can be unparsed to annotated source code.
The applicability of our approach was demonstrated by integrating into SATIrE the program analyzer generator PAG, the LLNL-ROSE source-to-source translator, and by generating an external representation of the ROSE-AST as Prolog term. We are using SATIrE [11] in a lecture on optimizing compilers at TU Vienna since 2006. Currently we focus on specifying different kinds of pointer analyses for evaluation with respect to scalability, WCET analyses, and on design pattern detection and extraction. Other tools that we are presently integrating are Stratego and iburg. Tools of interest to be integrated in future are model checking tools and automatic theorem provers.
We aim at providing a platform of integrated tools for program analysis research of multi-million line applications. We hope that the use of high-level specification languages permits a qualitative comparison of analyses and that the analysis and transformation of real-world application codes permits a quantitative evaluation of program analyses at a broad range in future.

Acknowledgements. This work has been funded in part by the ARTIST2 Network of Excellence (http://www.artist-embedded.org). I wish to thank Dan Quinlan for the cooperation and fruitful joint work on LLNL-ROSE, Florian Martin for the support in integrating

PAG, Adrian Prantl for his work on maintaining the Prolog term representation, Jens Knoop for his support in integrating SATIrE in various research projects, and all students who have contributed in several SATIrE projects: Gergo Barany, Viktor Pavlu, Christoph Bonitz.

References

1. U. Aßmann. How to uniformly specify program analysis and transformation with graph rewrite systems. In *Proceedings of the 6th International Conference on Compiler Construction* (*CC'96*) (*Linköping, Sweden*), Lecture Notes in Computer Science, vol. 1060, pages 121 – 135. Springer-Verlag, Heidelberg, Germany, 1996.
2. U. Aßmann. On edge addition rewrite systems and their relevance to program analysis. In *Proceedings of the 5th International Workshop on Graph Grammars and Their Application to Computer Science* (*GGTA'94*) (*Williamsburg*), Lecture Notes in Computer Science, vol. 1073, pages 321 – 335. Springer-Verlag, Heidelberg, Germany, 1996.
3. Edison Design Group. http://www.edg.com.
4. E. Glynn, I. Hayes, and A. MacDonald. Integration of generic program analysis tools into a software development environment. In *ACSC '05: Proceedings of the Twenty-eighth Australasian conference on Computer Science*, pages 249–257, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.
5. M. J. Harrold and G. Rothermel. Separate computation of alias information for reuse. *IEEE Trans. Softw. Eng.*, 22(7):442–460, 1996.
6. S. Lerner, T. Millstein, and C. Chambers. Automatically proving the correctness of compiler optimizations. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 220–231, New York, NY, USA, 2003. ACM Press.
7. F. Martin. PAG – an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.
8. A. Prantl. Source-to-source transformations for WCET analysis: The COSTA approach. In *24. Workshop der GI-Frachgruppe Programmiersprachen und Rechenkonzepte*, 2007.
9. D. Quinlan, M. Schordan, B. Miller, and M. Kowarschik. Parallel object-oriented framework optimization. *Concurrency and Computation: Practice and Experience*, 16, Issue 2-3:293–302, 2004.
10. M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, Jan. 1998.
11. SATIrE. http://www.complang.tuwien.ac.at/markus/satire. Static Analysis Tool Integration Engine.
12. M. Schordan and D. Quinlan. Specifying transformation sequences as computation on program fragments with an abstract attribute grammar. In *Proceedings of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05)*, pages 97–106. IEEE Computer Society Press, 2005.
13. D. Whitfield and M. L. Soffa. An approach for exploring code-improving transformations. *ACM Transactions on Programming Languages and Systems*, 19(6):1053 – 1084, 1997.