TOOL DEMONSTRATION

Cool: A Control-Flow Generator for System Analysis

Volker Braun[1], Jens Knoop[2], and Dirk Koschützki[2]

[1] Universität Dortmund, D-44221 Dortmund, Germany
e-mail: braun@ls5.cs.uni-dortmund.de
[2] Universität Passau, D-94030 Passau, Germany
e-mail: {knoop,koschuetzki}@fmi.uni-passau.de

Abstract Cool is a unifying control-flow analysis (CFA) generator for system analysis. It uniformly supports the automatic generation of transition systems and flow graphs from process algebra terms and programs of programming languages. Basically, it relies on "unrolling" its argument according to transition rules resembling structural operational semantic rules. As a side-effect of the unifying view of process algebra and programming language programs, Cool supports the automatic construction of CFA-components of optimizing compilers, which are usually still hand-coded. Thus, combining it with data-flow analysis and optimization generators like the DFA&OPT-METAFrame tool kit it renders possible the generation of complete optimizers.

1 Motivation and Overview

Procedures for system analysis and verification are typically designed for *automata*-like representations of the system under consideration. Two prominent examples are verification procedures for concurrent and distributed systems given in terms of *process algebra terms* (*programs*) and analysis procedures for *programs* of high-level *programming languages* for the generation of highly efficient code by optimizing compilers. In both cases the application of the relevant analysis and verification procedures relies on transforming the process algebra or programming language program into appropriate graphical representations, called *transition systems* and *flow graphs* in their respective contexts.

In the field of optimizing compilers this is accomplished by a *control-flow analysis* (*CFA*), which typically transforms the abstract syntax trees constructed by the parser into the corresponding *flow graphs*, which are the syntactic basis of the large majority of performance improving optimizations. Though CFA-components are thus a standard ingredient of optimizing compilers, they are usually still hand-coded. The short-comings are obvious: high expenditure, low portability, and costly extensibility. This situation is the more surprising as almost every other phase of compiler construction from lexical (cf. [8]) and syntactic analysis (cf. [5]) over data-flow analysis (cf. [1]) and optimization (cf. [9]) to

code generation (cf. [6]) is nowadays supported by powerful generators allowing their automatic construction from concise specifications.

In contrast, there have recently been proposed a number of successful approaches and tools based thereof for the automatic transfer of (CCS-like) process algebra programs into transition systems (cf. [4, 2]). In essence, the transformations realized by the *Process Algebra Compiler* (*PAC*) and the *Process Algebra Rewriting System* (*PARIS*) of [4] and [2] rely on the "unrolling" of the process algebra program (term) according to the *transition rules* of the *process algebra* under consideration as illustrated in Figure 1.

From the perspective of a compiler writer, this means interpreting the effect of communication as control flow. In fact, identifying programs of a process algebra with *programs* of a *programming language*, and transition rules of the process algebra with *rewriting rules* resembling the *structural operational semantic* (*SOS*) rules of the programming language, the construction principle becomes directly applicable to programming languages allowing the automatic transformation of programs into flow graphs as illustrated in Figure 2.

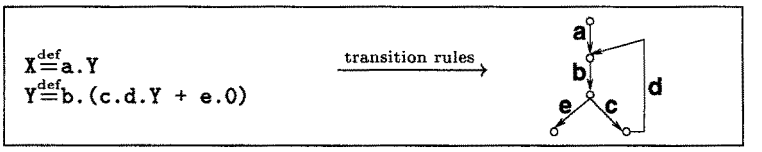

Figure 1. Transforming a process algebra term into a transition system.

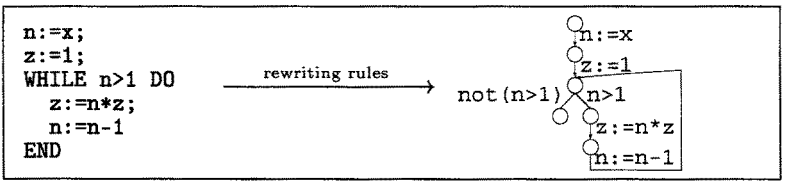

Figure 2. Transforming a program into a flow graph.

Exploiting this analogy systematically is the basis of our approach resulting in the CFA-generator Cool. Most closely related to it is the "Process Algebra Compiler" PAC presented in [4]. The application focus of both tools, however, and, as a consequence, the output generated is different. Whereas PAC yields a number of functions allowing the transformation of a process algebra term into a corresponding transition system according to the interface requirements of a variety of targeted verification tools, Cool provides by its unifying view of process algebras and programming languages an important contribution to the construction of optimizing compilers. Moreover, it supports the generation of user-customized graphs according to specific application-dependent requirements, which is both out of the scope of PAC. Simultaneously, this confirmatively answers a problem left for future research in [4] to which extent the SO-based construction principle can successfully be transferred and adapted to application scenarios different from that concentrated on in [4].

2 Screen Shots from a Sample Session

In this section we focus on the contribution of Cool for the construction of optimizing compilers. Here, the major benefits of our approach are as follows:

1. *Generality:* The full range of imperative and object-oriented languages is captured.
2. *Simplicity:* (i) Language extensions can modularly be captured by enlarging the current generator specification incrementally. (ii) The specification required for a new programming language can comprehensively be constructed in a "copy/paste"-style: adapting a specification at hand accordingly to the "syntactic sugar" of the new language suffices.
3. *Flexibility:* The structure of the graphs generated can easily be tailored according to application-specific requirements by adapting the transition rules of the specification.

All these features are discussed in detail in [3]. They are achieved by means of *concise specifications* consisting essentially of a set of SO_{CFA}-rules containing for every statement type (elementary and control statements) of the programming language considered a corresponding rewriting rule, which usually can be derived straightforwardly from the corresponding SOS-rule.

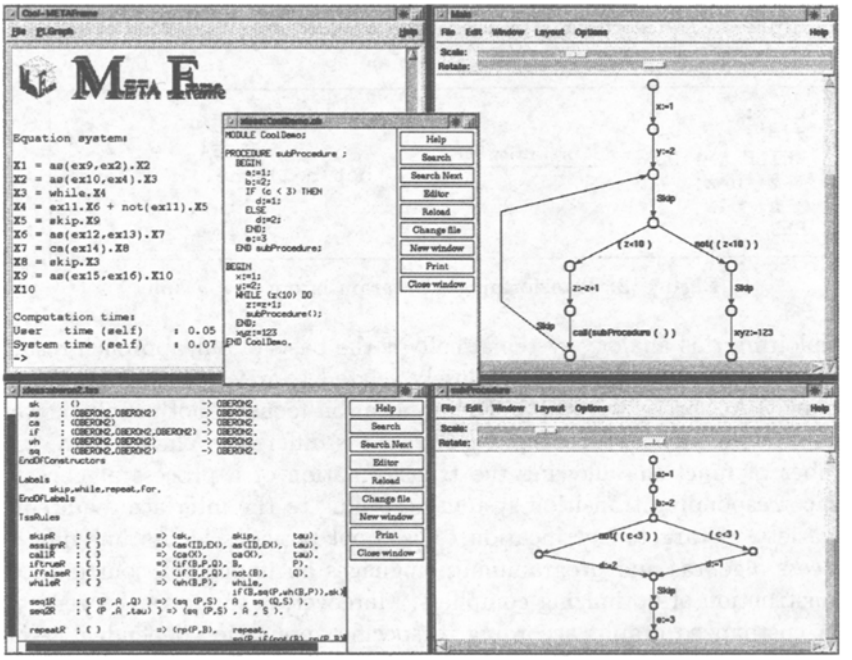

Figure 3. Cool: Screen-shot from a sample session.

This is illustrated in Figure 3 showing a snapshot from a sample session providing a flavour of the system. The upper left window shows the command shell of

the tool, while the lower left one displays a fragment of the SO_{CFA}-rule specification for Oberon-2. The windows on the right complement this presentation by showing the flow graphs generated by Cool for the program displayed in the central window by feeding its output into the automatical graph-layout component of the DFA&OPT-METAFrame system (cf. [10]).

3 Conclusions

Cool supports currently the automatic transfer of process algebra and programming language programs into transition systems and flow graphs, respectively, but it is not limited to these application scenarios. It has successfully been tested within the DFA&OPT-METAFrame project demonstrating that it captures the full range of imperative and object-oriented programming languages (cf. [7]). Currently, we are integrating Cool and DFA&OPT-METAFrame in order to arrive at CFA&DFA&OPT-METAFrame, a system which will be unique in supporting the construction of *complete* optimizers, i.e., of CFA- and DFA-components, and the optimizing transformations based thereof. The tool will be made available within the METAFrame system.

References

1. M. Alt and F. Martin. Generation of efficient interprocedural analyzers with PAG. In *Proc. 2nd Int. Static Analysis Symp. (SAS'95)*, LNCS 983, pages 33 – 50. Springer-V., 1995.
2. V. Braun. A transition system generator for process algebras. Master's thesis, RWTH Aachen, Germany, 1994. (In German).
3. V. Braun, J. Knoop, and D. Koschützki. Cool: A control-flow generator for system analysis. Technical Report MIP–9801, Fak. f. Math. u. Inf., Univ. Passau, Germany, 1998.
4. R. Cleaveland, E. Madelaine, and S. Sims. A front-end generator for verification tools. In *Proc. 1st Int. Workshop on Tools and Algorithms for Constr. and Analysis of Syst. (TACAS'95)*, LNCS 1019, pages 153 – 173. Springer-V., 1995.
5. Ch. Donnelly and R. M. Stallman. Bison, the YACC-compatible parser generator. Free Software Foundation, 1991.
6. C. W. Fraser and A. L. Wendt. Automatic generation of fast optimizing code generators. In *Proc. ACM SIGPLAN Conf. Prog. Lang. Design and Impl. (PLDI'88)*, volume *23,7* of *ACM SIGPLAN Not.*, pages 79 – 84, 1988.
7. M. Klein, J. Knoop, D. Koschützki, and B. Steffen. DFA&OPT-METAFrame: A tool kit for program analysis and optimization. In *Proc. 2nd Int. Workshop on Tools and Algorithms for Constr. and Analysis of Syst. (TACAS'96)*, LNCS 1055, pages 422 – 426. Springer-V., 1996.
8. G. T. Nicol. Flex, the lexical scanner generator. Free Software Foundation, 1993.
9. St. W. K. Tijan and J. L. Hennessy. Sharlit — A tool for building optimizers. In *Proc. ACM SIGPLAN Conf. Prog. Lang. Design and Impl. (PLDI'92)*, volume *27,7* of *ACM SIGPLAN Not.*, pages 82 – 93, 1992.
10. M. v. d. Beeck, V. Braun, A. Claßen, A. Dannecker, C. Friedrich, D. Koschützki, T. Margaria, F. Schreiber, and B. Steffen. Graphs in METAFrame: The unifying power of polymorphism. In *Proc. 3rd Int. Workshop on Tools and Algorithms for Constr. and Analysis of Syst. (TACAS'97)*, LNCS 1217, pages 112 – 129. Springer-V., 1997.