Data-Flow Analysis as Model Checking Within the jABC

Anna-Lena Lamprecht[1], Tiziana Margaria[1], and Bernhard Steffen[2]

[1] Service Engineering for Distributed Systems, Universität Göttingen, Germany
alamprec@stud.informatik.uni-goettingen.de, margaria@cs.uni-goettingen.de
[2] Chair of Programming Systems, Universität Dortmund, Germany
steffen@cs.uni-dortmund.de

Abstract. This paper describes how the jABC, a generic framework for library-based program development, and two of its plugins - the Model Checker and a flow graph converter - form a framework for intraprocedural data-flow analysis via model checking. Based on functionalities provided by the Soot program analysis platform, the converter generates graph structures from Java classes. Data flow analyses are then expressed as formulas in the modal μ-calculus. Executing the analysis is carried out by checking the validity of the formulas on the flow graph.

The tool demonstration will illustrate the interplay of the involved components, which elegantly provides a fully integrated implementation of Data-Flow Analysis as Model Checking in a software development environment.

1 Introduction

Static program analysis [8, 3, 1] aims at reliably approximating information about the actual run-time behavior of programs. It consists of two major steps: control-flow analysis, mainly used to generate a control flow graph that is used for the next step, and data-flow analysis (DFA), for collecting information about the program that might be of use for optimization or validation. Classical data-flow analyses use iterative algorithms, which compute a particular property for a given program, and can be characterized as follows [11, 10]:

DFA-algorithm for a property :
programs $\rightarrow$ program points satisfying the property

Model checking [2, 6], a technique for the automatic identification of states in a finite system that satisfy a specific modal or temporal formula, can be used for DFA with appropriate input [11, 10]:

model checker:
modal formulas $\times$ model $\rightarrow$ states satisfying the argument formula

Thus, if we have a model checker at our disposal and want to check a new program property, specifying a new DFA simply means writing a new formula.

A. Mycroft and A. Zeller (Eds.): CC 2006, LNCS 3923, pp. 101–104, 2006.

This is significantly different from traditional DFA specifications in terms of DFA frameworks or equational systems, as it allows to directly describe the desired outcome as temporal formula, rather than the way how it may be computed (in terms of transfer functions and the like).

During the tool demonstration we will illustrate the interplay of the involved components, which elegantly provides a fully integrated implementation of Data-Flow Analysis as Model Checking (DFA-MC) in a software development environment.

2 Data-Flow Analysis Via Model Checking

The connection between data-flow analysis and model checking described above implies what is required for DFA-MC [10, 11]: instead of programs we need models of programs, and instead of different DFA-algorithms for different properties we need a model checker and different modal formulas.

From Programs to Program Models. Slight variants of Kripke transition systems [6] work well for modeling sequential imperative programs for data flow analysis purposes, as they are able to concisely express the implied predicate transformer scheme [10, 11, 9].

Two variants are available in our framework. The first is closely related to the classical control flow graphs (CFGs), i.e. the nodes of the graph structure represent the statements and the predicates, while the edges represent the control flow (conditional or unconditional branching). The second, which we will use during our demonstration, is that the statements are pushed from the nodes into the outgoing edges. Thus, nodes express the predicates or results of the considered analysis, and edges labelled with the statements express the nodes' interdependencies. Pushing the statements downwards (i.e. into the outgoing edges), like it is done here, results in a *precondition* model.

Formally, a precondition program model P is a quintuple $(S, Act, \rightarrow, B, \lambda)$, where

1. S is a finite set of nodes or program states.
2. Act is a set of actions (i.e. the possible statements of the programming language).
3. $\rightarrow \subseteq S \times Act \times S$ is a set of labeled transitions, which define the control flow of P.
4. B is a set of atomic propositions.
5. λ is a function $\lambda : S \rightarrow 2^B$ that labels states with subsets of B.

Due to the similarities between control flow graphs and these program models, the latter can be generated analogously to the former. As we will see later, our framework generates its models directly from control flow graphs.

How to write Modal Specifications. In contrast to traditional DFA specifications in terms of DFA frameworks or equational systems, which specify *how* a certain

information about the program is computed (how specifications), DFA specifications via temporal formulas allow us to directly describe *what* we want to compute (what specifications). E.g., one describes a dead variable analysis directly as an analysis, which checks for a variable whether it is guaranteed to be redefined before any future use, rather than as a propagation of values keeping track of usage information (see [10, 11, 9] for details). This does not only simply the specification of new DFAs, but also simplifies the corresponding correctness and completeness proofs. Formally, these DFA specifications can intuitively be written in variations of CTL or in the modal mu calculus ([2]).

3 The jABC Framework

The Java-based *jABC* picks up the principles that have already been used in the C++-based Agent Building Center (ABC) since 1993 [4] and combines them with new ideas. The jABC is a commercial product as well as a student experimentation platform [7].

The typcial feature of the jABC framework is the usage of a graphical, high-level programming layer, where hierarchical directed graphs can be constructed from special components, called SIBs (Service Independent Building Blocks), that represent a unit of source code encoding a particular functionality. It is possible to define what programs do and in what order just by building such graphs from SIBs. Thus, an application designer does not necessarily have to have knowledge about "real" programming, provided some programming experts have developed an appropriate set of SIBs before.

Several plugins and extensions are available. Three of them, that form the main constituents of DFA-MC framework, are

- the **Model Checker**, which we will consider as a black box in the following, just as typical users of our method do,
- the **UnitGraph2SibGraph** plugin, which allows us to generate SIB-Graphs from Java-programs based on Soot functionality. Soot creates control flow graphs from Java source code, which can then be enriched by fetching information from the nodes of the CFGs in order to generate SIBGraphs, the input format for the Model Checker.
- the **Formula Builder**, which supports the convenient specification of temporal formulas. In particular, it supports the high-level specification of temporal properties, which are then translated into the mu-calculus.

Based on these ingredients, we were able to fully integrate the Data-Flow Analysis as Model Checking framework in the jABC software development environment. In this application of the jABC, SIBs are used in a particularly fine-granular form: we use a SIB per Jimple statement.

4 Conclusion

We illustrated our framework for the intraprocedural data-flow analysis of Java programs by means of model checking. Characteristic for our approach was the

underlying framework architecture of the jABC which allowed us to modularly realize the required functionality.

Currently, each formula has to be checked separately (including the adaption of the parameters by hand), making the analysis process quite arduous. We are therefore planning to develop a bit-vector functionality for the model checking plugin, and we are in the course of extending the model checker plugin to capture pushdown systems, which would directly provide us with the power of interprocedural analysis.

There are other plugins and extensions of the jABC which can be applied in the context of our DFA-MC framework. One example is the *jETI* (Java Electronic Tool Integration Platform) System [5], which can enable the framework to be executed remotely.

Acknowledgements. Many thanks to Marco Bakera, Clemens Renner and Marc Njoku for programming and technical support.

References

1. A. Aho and J. Ullman. *Principles of Compiler Design.*, volume 3. Addison-Wesley, 1979.
2. E. Clarke, O. Grumberg, and D. Peled. *Model Checking.*, volume 3. The MIT Press, 2001.
3. M. Hecht. *Flow Analysis of Computer Programs (Programming Languages Ser. Volume5).* Elsevier Science Ltd, 1977.
4. T. Margaria. Components, features, and agents in the abc. In M. Ryan, J. Meyer, and H. Ehrich, editors, *Objects, Agents, and Features*, volume 2975 of *Lecture Notes in Computer Science*, pages 154–174. Springer, 2003.
5. T. Margaria, R. Nagel, and B. Steffen. jeti: A tool for remote tool integration. In N. Halbwachs and L. Zuck, editors, *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 557–562. Springer, 2005.
6. M. Müller-Olm, D. Schmidt, and B. Steffen. Model-checking: A tutorial introduction. In A. Cortesi and G. Filé, editors, *SAS*, volume 1694 of *Lecture Notes in Computer Science*, pages 330–354. Springer, 1999.
7. R. Nagel. Java abc framework. http://jabc.cs.uni-dortmund.de, July 2005.
8. F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer, 1999.
9. D. Schmidt and B. Steffen. Program analysis *as* model checking of abstract interpretations. In G. Levi, editor, *SAS*, volume 1503 of *Lecture Notes in Computer Science*, pages 351–380. Springer, 1998.
10. B. Steffen. Data flow analysis as model checking. In T. Ito and A. Meyer, editors, *TACS*, volume 526 of *Lecture Notes in Computer Science*, pages 346–365. Springer, 1991.
11. B. Steffen. Generating data flow analysis algorithms from modal specifications. *Sci. Comput. Program.*, 21(2):115–139, 1993.