Bernhard Steffen
Giorgio Levi (Eds.)

LNCS 2937

Verification, Model Checking, and Abstract Interpretation

5th International Conference, VMCAI 2004
Venice, Italy, January 2004
Proceedings

Springer

Lecture Notes in Computer Science 2937

Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Hong Kong
London
Milan
Paris
Tokyo

Bernhard Steffen Giorgio Levi (Eds.)

Verification, Model Checking, and Abstract Interpretation

5th International Conference, VMCAI 2004
Venice, Italy, January 11-13, 2004
Proceedings

Springer

Volume Editors

Bernhard Steffen
Universität Dortmund, LS V
Baroper Str. 301, 44221 Dortmund, Germany
E-mail: steffen@cs.uni-dortmund.de

Giorgio Levi
Università di Pisa, Dipartimento di Informatica
Via Buonarroti, 2, 56100 Pisa, Italy
E-mail: levi@di.unipi.it

Cataloging-in-Publication Data applied for

A catalog record for this book is available from the Library of Congress.

Bibliographic information published by Die Deutsche Bibliothek
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data is available in the Internet at <http://dnb.ddb.de>.

CR Subject Classification (1998): F.3.1-2, D.3.1, D.2.4

ISSN 0302-9743
ISBN 3-540-20803-8 Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by PTP-Berlin, Protago-TeX-Production GmbH
Printed on acid-free paper SPIN: 10975695 06/3142 5 4 3 2 1 0

Preface

This volume contains the proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2004), held in Venice, January 11–13, 2004, in conjunction with POPL 2004, the 31st Annual Symposium on Principles of Programming Languages, January 14–16, 2004. The purpose of VMCAI is to provide a forum for researchers from three communities—verification, model checking, and abstract interpretation—which will facilitate interaction, cross-fertilization, and the advance of hybrid methods that combine the three areas. With the growing need for formal tools to reason about complex, infinite-state, and embedded systems, such hybrid methods are bound to be of great importance.

Topics covered by VMCAI include program verification, static analysis techniques, model checking, program certification, type systems, abstract domains, debugging techniques, compiler optimization, embedded systems, and formal analysis of security protocols.

This year's meeting follows the four previous events in Port Jefferson (1997), Pisa (1998), Venice (2002), LNCS 2294 and New York (2003), LNCS 2575. In particular, we thank VMCAI 2003's sponsor, the Courant Institute at New York University, for allowing us to apply a monetary surplus from the 2003 meeting to this one.

The program committee selected 22 papers out of 68 on the basis of three reviews. The principal criteria were relevance and quality. The program of VMCAI 2004 included, in addition to the research papers,

- a keynote speech by David Harel (Weizmann Institute, Israel) on *A Grand Challenge for Computing: Full Reactive Modeling of a Multicellular Animal*,
- an invited talk by Dawson Engler (Stanford University, USA) on *Static Analysis Versus Software Model Checking for Bug Finding*,
- an invited talk by Mooly Sagiv (Tel Aviv University, Israel) called *On the Expressive Power of Canonical Abstraction*, and
- a tutorial by Joshua D. Guttman (Mitre, USA) on *Security, Protocols, and Trust*.

We would like to thank the Program Committee members and the reviewers, without whose dedicated effort the conference would not have been possible. Our thanks go also to the Steering Committee members for helpful advice, to Agostino Cortesi, the Local Arrangements Chair, who also handled the conference's Web site, and to David Schmidt, whose expertise and support was invaluable for the budgeting. Special thanks are due to Martin Karusseit for installing, managing, and taking care of the METAFrame Online Conference Service, and to Claudia Herbers, who, together with Alfred Hofmann and his team at Springer-Verlag, collected the final versions and prepared the proceedings.

Special thanks are due to the institution that helped sponsor this event, the Department of Computer Science of Ca' Foscari University, and to the professional organizations that support the event: VMCAI 2004 is held in cooperation with ACM and is sponsored by EAPLS.

January 2004 Bernhard Steffen

Steering Committee

Agostino Cortesi (Italy)
E. Allen Emerson (USA)
Giorgio Levi (Italy)
Andreas Podelski (Germany)
Thomas W. Reps (USA)
David A. Schmidt (USA)
Lenore Zuck (USA)

Program Committee

Chairs: Giorgio Levi (University of Pisa)
Bernhard Steffen (Dortmund University)

Ralph Back (Åbo Akademi University, Finland)
Agostino Cortesi (Università Ca' Foscari di Venezia, Italy)
Radhia Cousot (CNRS and École Polytechnique, France)
Susanne Graf (VERIMAG Grenoble, France)
Radu Grosu (SUNY at Stony Brook, USA)
Orna Grumberg (Technion, Israel)
Gerhard Holzmann (Bell Laboratories, USA)
Yassine Lakhnech (Université Joseph Fourier, France)
Jim Larus (Microsoft Reseach, USA)
Markus Müller-Olm (FernUniversität in Hagen, Germany)
Hanne Riis Nielson (Technical University of Denmark, Denmark)
David A. Schmidt (Kansas State University, USA)
Lenore Zuck (New York University, USA)

Reviewers

Rajeev Alur
Roberto Bagnara
Ittai Balaban
Rudolf Berghammer
Chiara Bodei
Victor Bos
Dragan Bosnacki
Marius Bozga
Liana Bozga
Chiara Braghin
Roberto Bruni
Glenn Bruns
Sagar Chaki
Patrick Cousot
Silvia Crafa
Pierpaolo Degano
Benet Devereux
Agostino Dovier
Christian Ene
Javier Esparza
Jérôme Feret
Jean-Claude Fernandez
Gianluigi Ferrari
Riccardo Focardi
Martin Fränzle
John Gallagher
Roberto Giacobazzi
Arie Gurfinkel
Rene Rydhof Hansen
Jonathan Herzog
Patricia Hill
Frank Huch
Radu Iosif
Romain Janvier
Salvatore La Torre
Flavio Lerda
Francesca Levi
Flaminia Luccio
Jens Knoop
Daniel Kroening
Damiano Macedonio
Monika Maidl
Oded Maler
Damien Massé
Laurent Mauborgne
Fred Mesnard
Antoine Miné
Jean-François Monin
David Monniaux
Laurent Mounier
Kedar Namjoshi
Flemming Nielson
Sinha Nishant
Iulian Ober
Joël Ouaknine
Carla Piazza
Amir Pnueli
Cory Plock
Shaz Qadeer
Sriram Rajamani
Xavier Rival
Sabina Rossi
Grigore Rosu
Oliver Rüthing
Nicoletta Sabadini
Ursula Scheben
Axel Simon
Eli Singerman
Francesca Scozzari
Margaret H. Smith
Muralidhar Talupur
Simone Tini
Tayssir Touili
Stavros Tripakis
Enrico Tronci
Helmut Veith
Andreas Wolf
Ben Worrell
James Worrell
Aleksandr Zaks

Table of Contents

Formal Methods II

Software Checking

Invited Talk

Software Checking

Liveness and Completeness

Formal Methods III

Key Note

Preface

This volume contains the proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2004), held in Venice, January 11–13, 2004, in conjunction with POPL 2004, the 31st Annual Symposium on Principles of Programming Languages, January 14–16, 2004. The purpose of VMCAI is to provide a forum for researchers from three communities—verification, model checking, and abstract interpretation—which will facilitate interaction, cross-fertilization, and the advance of hybrid methods that combine the three areas. With the growing need for formal tools to reason about complex, infinite-state, and embedded systems, such hybrid methods are bound to be of great importance.

Topics covered by VMCAI include program verification, static analysis techniques, model checking, program certification, type systems, abstract domains, debugging techniques, compiler optimization, embedded systems, and formal analysis of security protocols.

This year's meeting follows the four previous events in Port Jefferson (1997), Pisa (1998), Venice (2002), LNCS 2294 and New York (2003), LNCS 2575. In particular, we thank VMCAI 2003's sponsor, the Courant Institute at New York University, for allowing us to apply a monetary surplus from the 2003 meeting to this one.

The program committee selected 22 papers out of 68 on the basis of three reviews. The principal criteria were relevance and quality. The program of VMCAI 2004 included, in addition to the research papers,

- a keynote speech by David Harel (Weizmann Institute, Israel) on *A Grand Challenge for Computing: Full Reactive Modeling of a Multicellular Animal*,
- an invited talk by Dawson Engler (Stanford University, USA) on *Static Analysis Versus Software Model Checking for Bug Finding*,
- an invited talk by Mooly Sagiv (Tel Aviv University, Israel) called *On the Expressive Power of Canonical Abstraction*, and
- a tutorial by Joshua D. Guttman (Mitre, USA) on *Security, Protocols, and Trust*.

We would like to thank the Program Committee members and the reviewers, without whose dedicated effort the conference would not have been possible. Our thanks go also to the Steering Committee members for helpful advice, to Agostino Cortesi, the Local Arrangements Chair, who also handled the conference's Web site, and to David Schmidt, whose expertise and support was invaluable for the budgeting. Special thanks are due to Martin Karusseit for installing, managing, and taking care of the METAFrame Online Conference Service, and to Claudia Herbers, who, together with Alfred Hofmann and his team at Springer-Verlag, collected the final versions and prepared the proceedings.

Special thanks are due to the institution that helped sponsor this event, the Department of Computer Science of Ca' Foscari University, and to the professional organizations that support the event: VMCAI 2004 is held in cooperation with ACM and is sponsored by EAPLS.

January 2004 Bernhard Steffen

Steering Committee

Agostino Cortesi (Italy)
E. Allen Emerson (USA)
Giorgio Levi (Italy)
Andreas Podelski (Germany)
Thomas W. Reps (USA)
David A. Schmidt (USA)
Lenore Zuck (USA)

Program Committee

Chairs: Giorgio Levi (University of Pisa)
Bernhard Steffen (Dortmund University)

Ralph Back (Åbo Akademi University, Finland)
Agostino Cortesi (Università Ca' Foscari di Venezia, Italy)
Radhia Cousot (CNRS and École Polytechnique, France)
Susanne Graf (VERIMAG Grenoble, France)
Radu Grosu (SUNY at Stony Brook, USA)
Orna Grumberg (Technion, Israel)
Gerhard Holzmann (Bell Laboratories, USA)
Yassine Lakhnech (Université Joseph Fourier, France)
Jim Larus (Microsoft Reseach, USA)
Markus Müller-Olm (FernUniversität in Hagen, Germany)
Hanne Riis Nielson (Technical University of Denmark, Denmark)
David A. Schmidt (Kansas State University, USA)
Lenore Zuck (New York University, USA)

Reviewers

Rajeev Alur
Roberto Bagnara
Ittai Balaban
Rudolf Berghammer
Chiara Bodei
Victor Bos
Dragan Bosnacki
Marius Bozga
Liana Bozga
Chiara Braghin
Roberto Bruni
Glenn Bruns
Sagar Chaki
Patrick Cousot
Silvia Crafa
Pierpaolo Degano
Benet Devereux
Agostino Dovier
Christian Ene
Javier Esparza
Jérôme Feret
Jean-Claude Fernandez
Gianluigi Ferrari
Riccardo Focardi
Martin Fränzle
John Gallagher
Roberto Giacobazzi
Arie Gurfinkel
Rene Rydhof Hansen
Jonathan Herzog
Patricia Hill
Frank Huch
Radu Iosif
Romain Janvier
Salvatore La Torre
Flavio Lerda
Francesca Levi
Flaminia Luccio
Jens Knoop
Daniel Kroening
Damiano Macedonio
Monika Maidl
Oded Maler
Damien Massé
Laurent Mauborgne
Fred Mesnard
Antoine Miné
Jean-François Monin
David Monniaux
Laurent Mounier
Kedar Namjoshi
Flemming Nielson
Sinha Nishant
Iulian Ober
Joël Ouaknine
Carla Piazza
Amir Pnueli
Cory Plock
Shaz Qadeer
Sriram Rajamani
Xavier Rival
Sabina Rossi
Grigore Rosu
Oliver Rüthing
Nicoletta Sabadini
Ursula Scheben
Axel Simon
Eli Singerman
Francesca Scozzari
Margaret H. Smith
Muralidhar Talupur
Simone Tini
Tayssir Touili
Stavros Tripakis
Enrico Tronci
Helmut Veith
Andreas Wolf
Ben Worrell
James Worrell
Aleksandr Zaks

Table of Contents

Formal Methods II

Software Checking

Invited Talk

Software Checking

Liveness and Completeness

Formal Methods III

Key Note

Security, Protocols, and Trust*

Joshua D. Guttman

guttman@mitre.org
http://www.ccs.neu.edu/home/guttman

Information security has benefited from mathematically cogent modeling and analysis, which can assure the absence of specific kinds of attacks. Information security provides the right sorts of problems: Correctness conditions may be subtle, but they have definite mathematical content. Systems may be complex, but the essential reasons for failures are already present in simple components. Thus, rigorous methods lead to clear improvements.

In this tutorial, we focus on one problem area, namely cryptographic protocols. Cryptographic protocols are often wrong, and we will start by studying how to break them. Most protocol failures arise from *unintended services* contained in the protocols themselves. An unintended service is an aspect of the protocol that requires legitimate principals unwittingly to provide an attacker with information that helps the attacker defeat the protocol. We describe a systematic way to discover unintended services and to piece them together into attacks.

Turning to the complementary problem of proving that there are no attacks on a particular protocol, we use the same insights to develop three basic patterns for protocol verification. These patterns concern the way that fresh, randomly chosen values ("nonces") are transmitted and later received back in cryptographically altered forms. We explain how these patterns, the *authentication tests*, are used to achieve authentication and to guarantee recency. They serve as a design method as well as a verification method.

In themselves, however, these methods do not explain the commitments that a principal makes by specific protocol actions, nor the trust one principal must have in another in order to be willing to continue a protocol run. In the last part of the tutorial, we describe how to combine protocol analysis with a *trust management logic* in order to formalize the trust consequences of executing protocols for electronic commerce and access control.

* Supported by the United States National Security Agency and the MITRE-Sponsored Research Program.

B. Steffen and G. Levi (Eds.): VMCAI 2004, LNCS 2937, p. 1, 2004.

Security Types Preserving Compilation*
(Extended Abstract)

Gilles Barthe[1], Amitabh Basu[2**], and Tamara Rezk[1]

[1] INRIA Sophia-Antipolis, France {Gilles.Barthe,Tamara.Rezk}@sophia.inria.fr
[2] IIT Delhi, India csu00099@cse.iitd.ernet.in

Abstract. Initiating from the seminal work of Volpano and Smith, there has been ample evidence that type systems may be used to enforce confidentiality of programs through non-interference. However, most type systems operate on high-level languages and calculi, and "low-level languages have not received much attention in studies of secure information flow" (Sabelfeld and Myers, [16]). Further, security type systems for low-level languages should appropriately relate to their counterparts for high-level languages; however, we are not aware of any study of type-preserving compilers for type systems for information flow.
In answer to these questions, we introduce a security type system for a low-level language featuring jumps and calls, and show that the type system enforces termination-insensitive non-interference. Then, we introduce a compiler from a high-level imperative programming language to our low-level language, and show that the compiler preserves security types.

1 Introduction

Type systems are popular artefacts to enforce safety properties in the context of mobile and embedded code. While such safety properties fail short of providing appropriate guarantees with respect to security policies to which mobile and embedded code must adhere, recent work has demonstrated that type systems are adequate to enforce statically security policies. These works generally focus on confidentiality and in particular on non-interference [7], which ensures confidentiality through the absence of information leakage. Initiating from the seminal work of Volpano, Smith and Irvine [20], type systems for non-interference have been thoroughly studied in the literature, see e.g. [16] for a survey. However, most works focus on high-level calculi, including λ-calculus, see e.g. [8], π-calculus, see e.g. [9], and ς-calculus [3], or high-level programming languages, including Java [2,12] and ML [15].

In contrast, relatively little is known about non-interference for low-level languages, in particular because their lack of structure renders control flow more intricate; in fact existing works, see e.g. [4,5], use model-checking and abstract

* Work partially supported by IST Projects Profundis and Verificard.
** This work was performed while the author was visiting INRIA Sophia-Antipolis.

B. Steffen and G. Levi (Eds.): VMCAI 2004, LNCS 2937, pp. 2–15, 2004.

interpretation techniques to detect illegal information flows, but do not provide proofs of non-interference for programs that are accepted by their analysis. Thus the first part of this paper is devoted to the definition of a security type system for a low-level language with jumps and calls, and a proof that the type system enforces termination-insensitive non-interference.

Of course, security type systems for low-level languages should appropriately relate to their counterparts for high-level languages. Indeed, one would expect that compilation preserves security typing. Thus the second part of the paper is devoted to a case study in compilation with security types: we define a high-level imperative language with procedures, and a compiler to the low-level language studied in the first part of the paper. Further, we endorse the language with a type system that guarantees termination-insensitive non-interference, and show that compilation function preserves typing. The proof that compilation preserves typing proceeds by induction on the structure of derivations, and can be viewed as a procedure to compute, from a certificate of well-typing at the source program, another certificate of well-typing for the compiled program. It is thus very close in spirit to a certifying compiler [13].

Contents. The remaining of the paper is organized as follows. In Section 2 we define an assembly language that shall serve as the compiler target, endorse it with a security type system, and prove that the type system enforces termination-insensitive non-interference. In Section 3, we introduce a high-level imperative language with procedures and its associated type system. Further, we introduce a compiler that we show to preserve security typing; we also discuss how type-preserving compilation can be used to lift non-interference to the high-level language. We conclude in Section 4, with related work and directions for further research.

2 Assembly Language

2.1 Syntax and Operational Semantics

The assembly language is a small language with jumps and procedures. A *program* P is a set of *procedures* with a distinguished, main, procedure; we let P_f be the procedure associated to an identifier f in P. Each procedure P_f consists of an array of instructions; we let $P_f[i]$ be the i-th instruction in P_f. The set Instr of instructions and the set Prog_c of compiled programs are defined in Figure 1. We often denote programs by $P_c :: [f := i^\star]^\star$. Given a program P, we let $\mathcal{PP}$ be its set of *programs points*, i.e. the set of pairs $\langle f, i\rangle$ with $f \in \mathcal{F}$, where $\mathcal{F}$ is a set of procedure names, and $i \in dom(P_f)$. Further, we assume programs to satisfy the usual well-formedness conditions, such as code containment: for every program point $\langle f, i\rangle$, $P_f[i] = \mathsf{if}\ j\ \Rightarrow\ j \in dom(P_f)$, etc.

The operational semantics is given as a transition relation between states. In our setting, values are integers, i.e. $\mathcal{V} = \mathbb{Z}$ and states are triples of the form $\langle \mathsf{cs}, \rho, s\rangle$ where $\mathsf{cs} \in \mathcal{PP}^\star$ is a *call string* whose length is bounded by some

$$
\begin{array}{rll}
i ::= & \mathsf{prim}\ op & \text{primitive value/operation} \\
| & \mathsf{load}\ x & \text{load value of } x \text{ on stack} \\
| & \mathsf{store}\ x & \text{store top of stack in } x \\
| & \mathsf{if}\ j & \text{conditional jump} \\
| & \mathsf{goto}\ j & \text{unconditional jump} \\
| & \mathsf{call}\ f & \text{procedure call} \\
| & \mathsf{return} & \text{return}
\end{array}
$$

where op is either a literal $n \in \mathbb{Z}$, or a primitive operation $+, -, \times, \ldots$, or a comparison operation $<, \leq, =, \ldots$; f ranges over a set $\mathcal{F}$ of procedure names, x ranges over a set $\mathcal{X}$ of registers, and j ranges over $\mathbb{N}$.

Fig. 1. Instruction set

previously agreed upon value max, $\rho : \mathcal{X} \to \mathcal{V}$ is a *register map* that assigns values to local variables (note that $\mathcal{X}$ is finite for any fixed program), and s is an *operand stack*, i.e. a stack of values. The operational semantics of the assembly language is given by the rules of Figure 2; all rules are subject to the proviso that the size of the call string and the operand stack remain bounded by some previously agreed upon maximal size max and MAX; further we assume given for every operation symbol op a corresponding total binary function $\underline{op}$ on integers. Finally, we write $\rho \oplus \{x \mapsto v\}$ to denote the unique function ρ' s.t. $\rho'(y) = \rho(y)$ if $y \neq x$ and $\rho'(x) = v$.

Observe that procedure calls do not activate a new frame with its own local variables and operand stacks, as e.g. in the JVM; in fact, procedures are closer to JVM subroutines than they are to JVM method invocations.

$$
\frac{P_f[i] = \mathsf{prim}\ n \qquad n \in \mathbb{Z}}{\langle\langle f, i\rangle :: \mathsf{cs}, \rho, s\rangle \rightsquigarrow \langle\langle f, i+1\rangle :: \mathsf{cs}, \rho, n :: s\rangle}
\qquad
\frac{P_f[i] = \mathsf{if}\ j \qquad v \neq 0}{\langle\langle f, i\rangle :: \mathsf{cs}, \rho, v :: s\rangle \rightsquigarrow \langle\langle f, j\rangle :: \mathsf{cs}, \rho, s\rangle}
$$

$$
\frac{P_f[i] = \mathsf{prim}\ op \qquad op \in \mathbb{O} \qquad n_1\ \underline{op}\ n_2 = n}{\langle\langle f, i\rangle :: \mathsf{cs}, \rho, n_1 :: n_2 :: s\rangle \rightsquigarrow \langle\langle f, i+1\rangle :: \mathsf{cs}, \rho, n :: s\rangle}
\qquad
\frac{P_f[i] = \mathsf{if}\ j \qquad v = 0}{\langle\langle f, i\rangle :: \mathsf{cs}, \rho, v :: s\rangle \rightsquigarrow \langle\langle f, i+1\rangle :: \mathsf{cs}, \rho, s\rangle}
$$

$$
\frac{P_f[i] = \mathsf{load}\ x}{\langle\langle f, i\rangle :: \mathsf{cs}, \rho, s\rangle \rightsquigarrow \langle\langle f, i+1\rangle :: \mathsf{cs}, \rho, \rho(x) :: s\rangle}
\qquad
\frac{P_f[i] = \mathsf{goto}\ j}{\langle\langle f, i\rangle :: \mathsf{cs}, \rho, s\rangle \rightsquigarrow \langle\langle f, j\rangle :: \mathsf{cs}, \rho, s\rangle}
$$

$$
\frac{P_f[i] = \mathsf{store}\ x}{\langle\langle f, i\rangle :: \mathsf{cs}, \rho, v :: s\rangle \rightsquigarrow \langle\langle f, i+1\rangle :: \mathsf{cs}, \rho \oplus \{x \mapsto v\}, s\rangle}
\qquad
\frac{P_{f'}[i] = \mathsf{call}\ f}{\langle\langle f', i\rangle :: \mathsf{cs}, \rho, s\rangle \rightsquigarrow \langle\langle f, 1\rangle :: \langle f', i\rangle :: \mathsf{cs}, \rho, s\rangle}
$$

$$
\frac{P_f[i] = \mathsf{return}}{\langle\langle f, i\rangle :: \langle f', i'\rangle :: \mathsf{cs}, \rho, s\rangle \rightsquigarrow \langle\langle f', i'+1\rangle :: \mathsf{cs}, \rho, s\rangle}
\qquad
\frac{P_f[i] = \mathsf{return}}{\langle\langle f, i\rangle :: \epsilon, \rho, s\rangle \rightsquigarrow \langle\epsilon, \rho, s\rangle}
$$

Fig. 2. Operational Semantics of Assembly Language

2.2 Defining Non-interference

Informally, non-interference guarantees that, executing a program P on initial states that are indistinguishable from the point of view of an attacker will not result in observable differences for the attacker. There are however a number of different ways in which this definition can be made precise, depending on the formulation of observability. One notion, that seems well adapted to our context, is termination-insensitive non-interference, which says that given any two states i and i', and assuming that executing P on i and i' respectively yield as final states f and f', indistinguishability between i and i' entails indistinguishability between f and f'. Note that such a definition implicitly assumes that an attacker cannot observe termination; there are stronger notions of confidentiality that consider termination and even execution time as observable, see e.g. [16]. Indistinguishability is a relation between states and is defined w.r.t. security maps that assign security levels to registers and to program points; throughout this paper, we assume that the set of security levels is $\mathcal{S} = \{H, L\}$ and is ordered by the clause $L \leq H$; considering a lattice of security levels instead is possible but adds technicalities without adding insight. Indistinguishability on states is defined in terms of indistinguishiblity relations on register maps, and on operand stacks. The former is a pointwise extension of the obvious indistinguishibility relation on values.

Definition 1 (Values and register maps indistinguishability).

- *Value indistinguishability $v \sim_{SL} v'$ (of values v and v' w.r.t. security level SL) is defined as $SL = H \vee v = v'$.*
- *The relation is extended pointwise to maps: for $\rho, \rho' : \mathcal{X} \to \mathcal{V}$ and $\Gamma : \mathcal{X} \to \mathcal{S}$, we have $\rho \sim_\Gamma \rho'$ is defined as*

$$\forall x \in \mathcal{X}.\ (\rho\ x) \sim_{(\Gamma\ x)} (\rho'\ x)$$

At this point, it is already possible to define non-interference for programs.

Definition 2. *A program P whose main procedure is* main *is* non-interferent *w.r.t. $\Gamma : \mathcal{X} \to \mathcal{S}$, written $\mathsf{NI}_\Gamma(P)$, if for every $\rho, \rho', \mu, \mu' : \mathcal{X} \to \mathcal{V}$ such that $\rho \sim_\Gamma \rho'$ and $\langle\langle \text{main}, 1\rangle, \rho, \epsilon\rangle \rightsquigarrow^\star \langle \epsilon, \mu, \epsilon\rangle$ and $\langle\langle \text{main}, 1\rangle, \rho', \epsilon\rangle \rightsquigarrow^\star \langle \epsilon, \mu', \epsilon\rangle$, we have $\mu \sim_\Gamma \mu'$.*

Stack indistinguishability requires a slight generalization of the pointwise order on stacks so as to handle high if with branches of different length(a motivation for this is given in section 2.4). The intuition is that we require operand stacks to be indistinguishable point-wise on some common top part, and then to be high in the bottom part on which they may not coincide. High operand stacks are defined relative to a stack type: formally, let s be an operand stack and st be a stack type; we write $\mathsf{high}_{os}\ (s, st)$ if s and st have the same length n and $st[i] = H$ for every $1 \leq i \leq n$.

Definition 3 (Operand stack indistinguishability). *Let s, s' be operand stacks and $st, st' \in \mathcal{ST}$. Then $s \sim_{st,st'} s'$ is defined inductively as follows:*

$$\frac{\mathsf{high}_o s\ (s, st) \qquad \mathsf{high}_o s\ (s', st')}{s \sim_{st,st'} s'} \qquad \frac{s \sim_{st,st'} s' \qquad v \sim_k v'}{v :: s \sim_{k::st,k::st'} v' :: s'}$$

Definition 4 (State indistinguishability). *State indistinguishability* $\sigma \sim_{\Gamma,st,st'} \sigma'$ *(of states* $\sigma = \langle \mathsf{cs}, \rho, s\rangle$ *and* $\sigma' = \langle \mathsf{cs}', \rho', s'\rangle$ *w.r.t. register type* Γ *and stack types* st *and* st'*) is defined as* $s \sim_{st,st'} s' \wedge \rho \sim_\Gamma \rho'$*.*

2.3 Control Dependence Regions

Type systems such as [18,20] reject programs that yield implicit flows through low assignments in a high branching instruction, e.g. $\mathsf{if}\ y_H\ \mathsf{then}\ x_L := 0\ \mathsf{else}\ x_L := 1$ that yield implicit flows through low assignments in a high branching instruction; in this program, the final value of x_L depends on y_H and thus the program is insecure.

In order to proceed likewise for our assembly language, we must resort to control dependence regions, which identify for every if instruction program points that execute under its control condition. While such control dependence regions are easy to identify in a structured, source language, their computation is slightly more intricate for unstructured assembly languages, but see e.g. [1] for algorithms that compute such regions.

For the purpose of this paper, we need not be precise about the exact computation of such control dependence regions. Instead, we define for every program P

$$\mathcal{PP}_{\mathrm{if}} = \{\langle f, i\rangle \in \mathcal{PP} \mid p_f[i] = \mathsf{if}\ j\}$$

and assume given two functions $\mathsf{reg} : \mathcal{PP}_{\mathrm{if}} \to \wp(\mathcal{PP})$ that computes the control dependence region of an if and $\mathsf{jun} : \mathcal{PP}_{\mathrm{if}} \to \mathcal{PP}$ that computes the junction point of the two branches of the if. Formally, we assume that

for every $\langle f', i'\rangle \in \mathcal{PP}_{\mathrm{if}}$, if $\langle f', i'\rangle \in \mathsf{reg}(\langle f, i\rangle)$ then $\mathsf{reg}(\langle f', i'\rangle) \subseteq \mathsf{reg}(\langle f, i\rangle)$—we latter refer to this property as RIP or region inclusion property—and that for every execution path

$$\langle\langle f, i\rangle :: \mathsf{cs}, \rho, s\rangle \rightsquigarrow \langle \mathsf{cs}_1, \rho_1, s_1\rangle \rightsquigarrow \ \ldots\ \rightsquigarrow \langle \mathsf{cs}_n, \rho_n, s_n\rangle$$

one of the following holds:

- $\mathsf{cs}_n = \langle f_s, i_s\rangle :: \ \ldots\ : \langle f_1, i_1\rangle :: \mathsf{cs}$ with $\langle f_k, i_k\rangle \in \mathsf{reg}(\langle f, i\rangle)$ for $1 \le k \le s$;
- there exists $1 \le l \le n$ such that $\mathsf{cs}_l = \mathsf{jun}(\langle f, i\rangle) :: \mathsf{cs}$;
- there exists $1 \le l \le n$ such that $\mathsf{cs}_l = \langle f, j\rangle :: \mathsf{cs}$ and $p_f[j] = \mathsf{return}$.

We shall use the function reg in the type system, and the assumption about execution paths in the proof of non-interference.

Remark. Strictly speaking, the function jun needs not be total. However, we can always extend jun to a total function with the required property by supposing that the last instruction of the main procedure is a return.

2.4 Type System

The type system is defined through an abstract transition system that manipulates stack types, and a security environment that associates to each program point a security level, and is parameterized by a register type $\Gamma : \mathcal{X} \to \mathcal{S}$ that sets the security level of each register. Before giving its formal definition, we motivate the type system on representative examples.

Example 1. Consider the following piece of program, where x_L is a low variable and y_H is a high variable:

$$\begin{array}{l} \mathsf{load}\ y_H \\ \mathsf{store}\ x_L \end{array}$$

The first instruction pushes the value held in y_H on top of the operand stack, while the second instruction stores the top of the operand stack in x_L. Thus this piece of program stores in the variable x_L the value held in the variable y_H, and yields a direct information leakage. Our type system prevents such explicit flows by restricting the transition of an instruction $\mathsf{store}\ x_L$ to the case where the type in the top of the stack type is low, whereas the instruction $\mathsf{load}\ y_H$ pushes a H in the stack type.

The security environment $se : \mathcal{PP} \to \mathcal{S}$ detects illicit flows by recording for each program point pp the highest security level of the control dependence influence under which the program point pp is.

Example 2. Consider the following pieces of program, where x_L is a low variable and y_H is a high variable:

1 load y_H	1 prim 0
2 if 6	2 store x_L
3 prim 0	3 load y_H
4 store x_L	4 if 6
5 goto 8	5 return
6 prim 1	6 prim 1
7 store x_L	7 store x_L
8 return	8 return

These pieces of program yield implicit flows, as a test on a high value yields different values for a low variable. Indeed, at program point 8, the low variable x_L contains the value 0 if y_H contains the value 0 or the value 1 if y_H contains a value different from 0. Our type system prevents such information leakage by imposing that the abstract transition rule for a high if instruction sets to H the security level of all program points in its control dependence region, and by rejecting low assignments and returns—unless the return is the last instruction of the procedure being executed— that are performed at high program points.

The abstract transition system is defined by the typing transfer rules in Figure 3. These rules, which capture the transformation of security types information by instructions, are of the form

$$\frac{\mathsf{cs} = \langle f, i\rangle :: \mathsf{cs}_0 \quad P_f[i] = \text{instruction}}{\Gamma, \mathsf{cs} \vdash st, se \Rightarrow st', se'}$$

where Γ is a fixed register type, and st, se determines typing constraints for cs, and st', se' determines typing constraints for the successors of cs. Note the transfer function rules define a partial function, i.e. $\Gamma, \mathsf{cs} \vdash st, se \Rightarrow st_1, se_1$ and $\Gamma, \mathsf{cs} \vdash st, se \Rightarrow st_2, se_2$ implies $st_1 = st_2$ and $se_1 = se_2$.

The successor relation $\mapsto \subseteq \mathcal{CS} \times \mathcal{CS}$, where $\mathcal{CS}$ is the set of $\mathcal{PP}$-lists of length $< \mathsf{max}$, is defined by the clauses:

- if $p_f[i] = \mathsf{return}$ then $\langle f, i\rangle :: \langle f', j\rangle :: \mathsf{cs}' \mapsto \langle f', j+1\rangle :: \mathsf{cs}'$ and $\langle f, i\rangle :: \epsilon \mapsto \epsilon$;
- if $p_f[i] = \mathsf{call}\ f'$ then $\langle f, i\rangle :: \mathsf{cs} \mapsto \langle f', 1\rangle :: \langle f, i\rangle :: \mathsf{cs}$;
- if $p_f[i] = \mathsf{goto}\ j$ then $\langle f, i\rangle :: \mathsf{cs} \mapsto \langle f, j\rangle :: \mathsf{cs}$;
- if $p_f[i] = \mathsf{if}\ j$ then $\langle f, i\rangle :: \mathsf{cs} \mapsto \langle f, k\rangle :: \mathsf{cs}$ for $k \in \{i+1, j\}$;
- otherwise, $\langle f, i\rangle :: \mathsf{cs} \mapsto \langle f, i+1\rangle :: \mathsf{cs}$.

Type-checking is performed by a dataflow analysis that explores all abstract execution paths; following Brisset and Coglio [6], we opt for a polyvariant analysis. Hence our type system deals with security types of the form $\mathcal{CS} \to \wp(\mathcal{ST} \times \mathcal{SE})$, where the set $\mathcal{ST}$ of stack types is defined as the set of $\mathcal{S}$-stacks of length smaller than MAX, and the set $\mathcal{SE}$ of security environments is defined as $\mathcal{PP} \to \mathcal{S}$; given a security type S and a call string cs we let S_{cs} denote $S(\mathsf{cs})$. For the purpose of this paper, we work with judgments of the form $\Gamma, S, \mathsf{cs} \vdash P$ where S is a security type; the typing rule is

$$\frac{\forall st, se \in S_{\mathsf{cs}}.\ \forall \mathsf{cs}' \in \mathcal{CS}.\ \mathsf{cs} \mapsto \mathsf{cs}' \ \Rightarrow\ \exists st', se' \in S_{\mathsf{cs}'}.\ \Gamma, \mathsf{cs} \vdash st, se \Rightarrow st', se'}{\Gamma, S, \mathsf{cs} \vdash P}$$

(There are standard algorithms to compute S when it exists, see e.g. [11]). Finally, we say that a program P has type S w.r.t. Γ, written $\Gamma, S \vdash P$, if $\Gamma, S, \mathsf{cs} \vdash p$ for all cs s.t. $\langle \mathsf{main}, 1\rangle :: \epsilon \mapsto^\star \mathsf{cs}$, where $\mapsto^\star$ denotes the transitive closure of $\mapsto$. As usual, we say that P is typable w.r.t. Γ, written $\Gamma \vdash P$, if $\Gamma, S \vdash P$ for some S.

2.5 Soundness

Typable programs are non-interferent.

Theorem 1. *If* $\Gamma \vdash P$ *then* $\mathsf{NI}_\Gamma(P)$.

The idea of the proof is as follows: first, we prove in Lemma 1 that indistinguishability is preserved under one step of execution, if the program is typable. Second, we prove in Lemma 2 that one step execution in a high-level environment yields a result state that is indistinguishable from the original one. By combining these results together, we conclude.

In the sequel, we use $s \cdot \mathsf{cs}$ to denote the call string of a state s and hd to denote the head function. We also write $\mathsf{high}\ st$ if all elements in the $\mathcal{S}$-stack s are high. We also write $\Gamma \vdash \mathsf{cs}, st, se \Rightarrow \mathsf{cs}', st', se'$ if $\Gamma, \mathsf{cs} \vdash st, se \Rightarrow st', se'$ and $\mathsf{cs} \mapsto \mathsf{cs}'$, and use $\Gamma \vdash \cdot, \cdot, \cdot \Rightarrow^\star \cdot, \cdot, \cdot$ to denote its transitive closure.

$$\frac{\mathsf{cs} = \langle f, i\rangle :: \mathsf{cs}' \quad P_f[i] = \mathsf{load}\ x}{\Gamma, \mathsf{cs} \vdash st, se \Rightarrow (\Gamma(x) \sqcup se(f,i)) :: st, se}$$

$$\frac{\mathsf{cs} = \langle f, i\rangle :: \mathsf{cs}' \quad P_f[i] = \mathsf{store}\ x \quad k \sqcup se(i) \leq \Gamma(x)}{\Gamma, \mathsf{cs} \vdash k :: st, se \Rightarrow st, se}$$

$$\frac{\mathsf{cs} = \langle f, i\rangle :: \mathsf{cs}' \quad P_f[i] = \mathsf{prim}\ n}{\Gamma, \mathsf{cs} \vdash st, se \Rightarrow se(f,i) :: st, se}$$

$$\frac{\mathsf{cs} = \langle f, i\rangle :: \mathsf{cs}' \quad P_f[i] = \mathsf{prim}\ op}{\Gamma, \mathsf{cs} \vdash k_1 :: k_2 :: st, se \Rightarrow (k_1 \sqcup k_2 \sqcup se(f,i)) :: st, se}$$

$$\frac{\mathsf{cs} = \langle f, i\rangle :: \mathsf{cs}' \quad P_f[i] = \mathsf{if}\ j}{\Gamma, \mathsf{cs} \vdash k :: st, se \Rightarrow \mathrm{lift}_k(st), \mathrm{lift}_k(se, \mathsf{reg}(\langle f, i\rangle))}$$

$$\frac{\mathsf{cs} = \langle f, i\rangle :: \mathsf{cs}' \quad P_f[i] = \mathsf{return} \quad se(f,i) = L \vee i + 1 \notin dom(P_f)}{\Gamma, \mathsf{cs} \vdash st, se \Rightarrow st, se}$$

$$\frac{\mathsf{cs} = \langle f, i\rangle :: \mathsf{cs}' \quad P_f[i] \in \{\mathsf{goto}\ j, \mathsf{call}\ f'\}}{\Gamma, \mathsf{cs} \vdash st, se \Rightarrow st, se}$$

where

- $\sqcup$ denotes the lub of two security levels;
- $\mathrm{lift}_k(st)$, where $k \in \mathcal{S}$, denotes the pointwise extension to the stack type st of the function $\lambda l.\ k \sqcup l$;
- $\mathrm{lift}_k(se, R)$, where $k \in \mathcal{S}$, denotes the pointwise extension for all program points in R of the function $\lambda l.\ k \sqcup l$.

Fig. 3. Transfer rules for instructions

Lemma 1 (One-Step Noninterference in Low-Level Environments). *Suppose* $\Gamma, S \vdash P$. *Let* s_1, s_2, s_1', s_2' *be states with* $s_1 \cdot \mathsf{cs} = s_2 \cdot \mathsf{cs}$ *and let* $(st_1, se), (st_2, se) \in S_{s_1 \cdot \mathsf{cs}}$ *be security types s.t.* $s_1 \sim_{\Gamma, st_1, st_2} s_2$, *and* $s_1 \rightsquigarrow s_1'$, *and* $s_2 \rightsquigarrow s_2'$.

Then there exist $(st_1', se_1') \in S_{s_1' \cdot \mathsf{cs}}$ *and* $(st_2', se_2') \in S_{s_2' \cdot \mathsf{cs}}$ *s.t.* $s_1' \sim_{\Gamma, st_1', st_2'} s_2'$. *Furthermore, one of the following holds:*

- $se_1' = se_2' = se$ *and* $s_1' \cdot \mathsf{cs} = s_2' \cdot \mathsf{cs}$;
- $s_1 \cdot \mathsf{cs} = \langle f, i\rangle :: \mathsf{cs}'$ *and* $P_f[i] = \mathsf{if}\ j$ *and* $\mathsf{hd}\ st_1 = \mathsf{hd}\ st_2 = H$.

Proof. By a case analysis on the instruction that is executed.

Lemma 2 (One-Step Noninterference in High-Level Environments). *Suppose* $\Gamma, S \vdash P$. *Let* s, s' *be states and* $(st, se) \in S_{s \cdot \mathsf{cs}}$ *be a security type s.t.* $s \cdot \mathsf{cs} = \langle f, i\rangle :: \mathsf{cs}'$, $s \rightsquigarrow s'$, $\mathsf{high}\ st$, *and* $se(\langle f, i\rangle) = H$; *Then there exists* $(st', se') \in S_{s' \cdot \mathsf{cs}}$ *s.t.* $\mathsf{high}\ st'$, *and* $s \sim_{\Gamma, st, st'} s'$, *and* $\Gamma, s \cdot \mathsf{cs} \vdash (st, se) \Rightarrow (st', se')$.

Proof. By a case analysis on the instruction that is executed.

Proof (of Theorem 1). Consider the two execution paths

$$\begin{array}{l} s_0 \rightsquigarrow s_1 \rightsquigarrow \ \ldots \ \rightsquigarrow s_{n_1} \\ s_0' \rightsquigarrow s_1' \rightsquigarrow \ \ldots \ \rightsquigarrow s_{n_2}' \end{array}$$

where $s_0 = \langle\langle \text{main}, 1\rangle :: \epsilon, \rho, \epsilon\rangle$, and $s'_0 = \langle\langle \text{main}, 1\rangle :: \epsilon, \rho', \epsilon\rangle$, and $s_{n_1} \cdot \mathsf{cs} = s_{n_2} \cdot \mathsf{cs} = \epsilon$.

By invoking Lemma 1 as long as it applies, we conclude for some maximal q that $s_q \cdot \mathsf{cs} = s'_q \cdot \mathsf{cs}$, and that there exists $(st_q, se), (st'_q, se) \in S_{s_q \cdot \mathsf{cs}}$ such that $s'_q \sim_{\Gamma, st_q, st'_q} s'_q$. Now there are two cases to treat: if $s_q \cdot \mathsf{cs} = \epsilon$ then $n_1 = n_2 = q$ and we are done; otherwise, the last instruction executed is an if high. By the typing rule of the if instruction, it holds high st_q and high st'_q. We now invoke Lemma 2 repeatedly to conclude that there exists s_{q_1} and s'_{q_2} and $(st_{q_1}, se_1) \in S_{s_{q_1} \cdot \mathsf{cs}}$ and $(st'_{q_2}, se'_2) \in S_{s'_{q_2} \cdot \mathsf{cs}}$ such that $s_q \sim_{\Gamma, st_q, st_{q_1}} s_{q_1}$ and $s'_q \sim_{\Gamma, st'_q, st'_{q_2}} s'_{q_2}$. By transitivity, we conclude $s_{q_1} \sim_{\Gamma, st_{q_1}, st'_{q_2}} s'_{q_2}$. Further, we can choose q_1 and q_2 to be the minimal indexes such that $s_{q_1} \cdot \mathsf{cs} = \mathsf{jun}(\langle f, i\rangle) :: \mathsf{cs}'$ and $s_{q_2} \cdot \mathsf{cs} = \mathsf{jun}(\langle f, i\rangle) :: \mathsf{cs}'$ respectively. As $\Gamma \vdash s_q \cdot \mathsf{cs}, st_q, se \Rightarrow^\star s_{q_1} \cdot \mathsf{cs}, st_{q_1}, se_1$ and $\Gamma \vdash s_q \cdot \mathsf{cs}, st'_q, se' \Rightarrow^\star s'_{q_2} \cdot \mathsf{cs}, st'_{q_2}, se'_2$ and only if statements may modify the security environment, and we assume RIP, we can further conclude that $se_1 = se_2$.

Thus we can apply Lemma 1 again, and repeat the process until reaching the final states of the reduction sequences.

3 Security Type Preserving Compilation

In this section, we define a high-level imperative language, endorse it with a security type system, and introduce a compiler from the source language to the assembly language. Then we show that the compiler preserves security types, and derive as a corollary that the security type system for the source language enforces non-interference.

3.1 Source Language

The source language is a simple imperative language with procedures. A procedure is a declaration of the form $\mathsf{proc}\ f(\boldsymbol{x}) = c;\ \mathsf{return}$ where f is a procedure name and c is a command. As with the assembly language, we assume that a program is a list of procedures with a distinguished, main, procedure without parameters. Formally, the set Expr of *expressions*, Comm of *commands*, and Prog of *programs* are given by the following syntaxes:

$$
\begin{aligned}
e &::= x \mid n \mid e\ op\ e \\
c &::= x := e \mid f\ (\boldsymbol{e}) \mid c; c \mid \mathsf{while}\ e\ \mathsf{do}\ c \mid \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c \\
P &::= [\mathsf{proc}\ f(\boldsymbol{x}) = c;\ \mathsf{return}]^\star
\end{aligned}
$$

The operational, big-step semantics of programs is based on judgments of the form $\langle c, \mu\rangle \Longrightarrow \mu'$, where $c \in \mathsf{Comm}$ and $\mu, \mu' : \mathcal{X} \to \mathcal{V}$. Rules are standard, see e.g. [18,20], and omitted. The security type system is based on judgments of the form $\Gamma \vdash_S e : \tau$ and $\Gamma \vdash_S c : \tau\ cmd$. A program P is typable, written $\Gamma \vdash_S P$, if $\Gamma \vdash_S \text{main} : \tau\ cmd$ for some τ. The typing rules are inspired from [18,

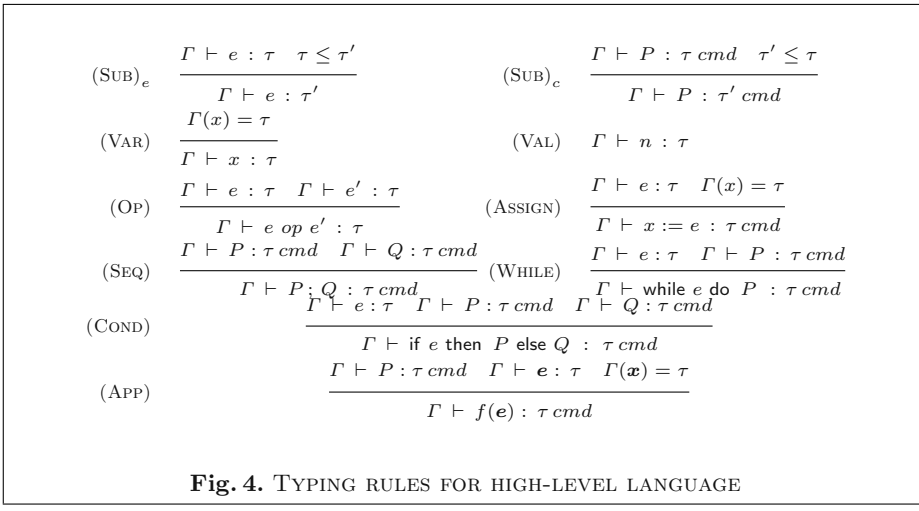


Fig. 4. Typing rules for high-level language

20], and are given in Figure [1] 4; in the last rule, we assume that the procedure f is defined by $\mathsf{proc}\ f\ (\boldsymbol{x}) = P;\ \mathsf{return}$. The typing rules exclude mutual and self-recursion; however it is possible to overcome this limitation at the price of further technicalities.

3.2 Compilation

The compilation function $\mathcal{C}_p : \mathsf{Prog} \to \mathsf{Prog}_c$ is defined in the usual way from a compilation function on expressions $\mathcal{C}_e : \mathsf{Expr} \to \mathsf{Instr}^\star$, and a compilation function on commands $\mathcal{C}_c : \mathsf{Comm} \to \mathsf{Instr}^\star$. Their formal definitions are given in Figure 5. In order to enhance readability, we use :: both for consing an element to a list and concatenating two lists, and we omit details of calculating pc in the clauses for while and if expressions. We also use $\#l$ to denote the length of a list.

3.3 Preservation of Security Types

Compilation preserves typing.

Theorem 2. *If $\Gamma \vdash_S P$ then $\Gamma \vdash \mathcal{C}_p(P)$.*

The proof proceeds in two steps. First, we show how to compute from an expression in the source language and its type, the type of the corresponding compiled code produced by the function $\mathcal{C}_e$. By abuse of notation, we write $se = \tau$ if $se(\langle f, j\rangle) = \tau$ for every $\langle f, j\rangle \in \mathcal{PP}$.

Lemma 3. *Assume e is an expression in P and $\Gamma \vdash_S e : \tau$, and $\mathcal{C}_p(P)_f[i \ldots j] = \mathcal{C}_e(e)$. For every $\mathsf{cs}_0 \in \mathcal{CS}$ and $st, se \in \mathcal{ST}$ s.t. $se = \tau$, there exists $S^e_{\mathsf{cs}_0,st,se} : \{\langle f, k\rangle :: \mathsf{cs}_0 \mid i \leq k \leq j+1\} \to \mathcal{ST}$—by abuse of notation, we often write S^e—s.t.:*

[1] For readability, we write $\vdash$ for $\vdash_S$.

$$
\begin{aligned}
\mathcal{C}_e(x) &= \mathsf{load}\ x \\
\mathcal{C}_e(n) &= \mathsf{prim}\ n \\
\mathcal{C}_e(e\ op\ e') &= \mathcal{C}_e(e) :: \mathcal{C}_e(e') :: \mathsf{prim}\ op \\
\mathcal{C}_c(x := e) &= \mathcal{C}_e(e) :: \mathsf{store}\ x \\
\mathcal{C}_c(f\ \boldsymbol{e}) &= \mathcal{C}_e(\boldsymbol{e}) :: \mathsf{call}\ f \\
\mathcal{C}_c(c_1;\ c_2) &= \mathcal{C}_e(c_1) :: \mathcal{C}_e(c_2) \\
\mathcal{C}_c(\mathsf{while}\ e\ \mathsf{do}\ c) &= \mathit{let}\ l_1 = \mathcal{C}_e(e); l_2 = \mathcal{C}_c(c); x = \#l_2; y = \#l_1\ \mathit{in} \\
&\qquad \mathsf{goto}\ (pc + x + 1) :: l_2 :: l_1 :: \mathsf{if}\ (pc - x - y) \\
\mathcal{C}_c(\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) &= \mathit{let}\ l_e = \mathcal{C}_e(e); lc_1 = \mathcal{C}_c(c_1); lc_2 = \mathcal{C}_c(c_2); \\
&\qquad x = \#lc_2; y = \#lc_1\ \mathit{in} \\
&\qquad l_e :: \mathsf{if}\ (pc + x + 2) :: lc_2 :: \mathsf{goto}\ (pc + y + 1) :: lc_1 \\
\mathcal{C}_p[\mathsf{proc}\ f(\boldsymbol{x}) := c;\ \mathsf{return}]^\star &= [f := (\mathsf{store}\ \boldsymbol{x} :: \mathcal{C}_c(c) :: \mathsf{return})]^\star
\end{aligned}
$$

Fig. 5. Compilation of expressions and commands

1. *for every* $\mathsf{cs}, \mathsf{cs}' \in dom(S^e)$*, if* $\mathsf{cs} \mapsto \mathsf{cs}'$ *then* $\Gamma, \mathsf{cs} \vdash S^e(\mathsf{cs}) \Rightarrow S^e(\mathsf{cs}')$*;*
2. $S^e(\langle f, j+1\rangle :: \mathsf{cs}_0) = \tau :: st, se$*.*

Proof. By structural induction on instructions.

Second, we extend the result to commands.

Lemma 4. *Assume* c *is a command in* P*, and* $\Gamma \vdash_S c : \tau\, cmd$*, and* $\mathcal{C}_p(P)_f[i \ldots j] = \mathcal{C}_c(c)$*. For every* $\mathsf{cs}_0 \in \mathcal{CS}$ *and* $st_0, se_0 \in \mathcal{ST}$ *s.t.* $se_0 = \tau$*, there exists* $S^c_{\mathsf{cs}_0, st_0, se_0} : \mathcal{CS} \rightharpoonup \wp(\mathcal{ST})$*—by abuse of notation, we often write* S^c*—s.t.:*

1. *for every* $\mathsf{cs}, \mathsf{cs}' \in dom(S^c)$ *and* $st, se \in S^c(\mathsf{cs})$ *s.t.* $\mathsf{cs} \mapsto \mathsf{cs}'$*, there exists* $st', se' \in S^c(\mathsf{cs}')$ *s.t.* $\Gamma, \mathsf{cs} \vdash st, se \Rightarrow st', se'$*;*
2. *there exists* st' *s.t.* $st' = st_0$ *or* $st' = \mathsf{lift}_\tau\ st_0$*, and for every* $\mathsf{cs} \in dom(S^c)$*,* $\mathsf{cs}' \notin dom(S^c)$ *and* $st, se \in S^c(\mathsf{cs})$ *s.t.* $\mathsf{cs} \mapsto \mathsf{cs}'$*,* $\Gamma, \mathsf{cs} \vdash st, se \Rightarrow st', se_0$*. We write* $\phi^c_{\mathsf{cs}_0, st_0, se_0}$ *for* st'*.*

Proof. By structural induction on instructions.

Proof (of Theorem 2). Set $se_0 = \tau$. By construction, the function $S^{c_{\mathrm{main}}}_{\langle \mathrm{main}, 1\rangle : \epsilon, \epsilon, se_0}$ is defined for all cs s.t. $\langle \mathrm{main}, 1\rangle : \epsilon \mapsto^\star \mathsf{cs}$. It is then immediate to conclude.

3.4 Recovering Non-interference for the Source Language

One can also prove that compilation preserves operational semantics.

Proposition 1 (Preservation of semantics). *Let* p *be a program whose main procedure is* $[\mathrm{main} := c_{\mathrm{main}}; \mathsf{return}]$ *and* $\rho : \mathcal{X} \to \mathcal{V}$*. If* $\langle c_{\mathrm{main}}, \rho\rangle \Longrightarrow \mu$ *then the compiled program satisfies* $\langle \langle \mathrm{main}, 1\rangle, \rho, \epsilon\rangle \rightsquigarrow^\star \langle \epsilon, \mu, \epsilon\rangle$*.*

By combining Proposition 1 and Theorem 2 we are able to recover the non-interference result for typable source programs.

Corollary 1 (Non-interference for source language). *Let P be a program, let $\Gamma : \mathcal{X} \to \mathcal{S}$ and assume that $\Gamma \vdash_S P$. Then P is non-interferent w.r.t. Γ in the sense that for every $\rho, \rho', \mu, \mu' : \mathcal{X} \to \mathcal{V}$,*

$$(\rho \sim_\Gamma \rho' \wedge \langle c_{\text{main}}, \rho\rangle \Longrightarrow \mu \wedge \langle c_{\text{main}}, \rho'\rangle \Longrightarrow \mu') \Longrightarrow \mu \sim_\Gamma \mu'$$

Proof. By Proposition 1, $\langle\langle \text{main}, 1\rangle, \rho, \epsilon\rangle \rightsquigarrow^\star \langle \epsilon, \mu, \epsilon\rangle$ and $\langle\langle \text{main}, 1\rangle, \rho', \epsilon\rangle \rightsquigarrow^\star \langle \epsilon, \mu', \epsilon\rangle$. Furthermore $\Gamma \vdash \mathcal{C}_p(P)$ by Theorem 2. Hence $\mathcal{C}_p(P)$ is non-interferent w.r.t. Γ by Theorem 1, and thus $\mu \sim_\Gamma \mu'$ by definition of non-interference.

4 Conclusion

We have shown how type systems can be used to enforce non-interference in a low-level language with procedures, and that one can define a security types preserving compiler from a high-level imperative language to such a low-level language.

4.1 Related Work

As emphasized in the introduction, static enforcement of non-interference through type systems is a well-researched topic, see e.g. [16] for a survey, and we can only comment on some of the most relevant literature.

Procedures and exceptions. Non-interference for procedures and exceptions has first been studied (for high-level languages) by Volpano and Smith [19,18]. Improved type systems for exceptions have been studied by Myers for Java [12] and by Pottier and Simonet for ML [15,17].

Low-level languages. Lanet *et al.*, see e.g. [5], develop a method to detect illicit flows for a sequential fragment of the JVM. In a nutshell, they proceed by specifying in the SMV model checker a symbolic transition semantics of the JVM that manipulates security levels, and by verifying that an invariant that captures the absence of illicit flows is maintained throughout the (abstract) program execution. Their analysis is more flexible than ours, in that it accepts programs such as $y_L := x_H; y_L := 0$. However, they do not provide a proof of non-interference. The approach of Lanet *et al.* has been refined by Bernardeschi and De Francesco, see e.g. [4], for a subset of the JVM that includes jumps, subroutines but no exceptions.

Type preserving compilation. Type preserving compilation has been thoroughly studied in the context of typed intermediate languages, most notably for ML and Java, see e.g. [10]. Information flow types preserving compilation has been studied by Zwandewic and Myers in the context of λ-calculus and CPS translation [21]. Also, Honda and Yoshida [9] consider type-preserving interpretations of higher-order imperative calculi with security types to π-calculus with security types. A similar result for resource control is being pursued in the MRG project[2].

Certifying compilation, as advocated by Proof Carrying Code [13], extends the idea of type preserving compilation by producing, from a certificate (i.e. a proof object) that a source program adheres to a property, a certificate that the compiled program adheres to a corresponding property, see e.g. [14].

4.2 Future Work

Our work constitutes a preliminary investigation in the realm of certifying compilation for security properties, and may be extended in several directions.

- Language expressiveness: we would like to extend the results of this paper to more powerful languages that include objects and/or higher-order functions. We are particularly interested in scaling up our results to the sequential fragment of Java and of the JVM, building up on [2] for the former and on recent, unpublished, work by the authors for the latter.
- Generality: the main result of this paper is specialized to one particular compilation function that departs from standard compilers, e.g. by not being optimizing. We would like to isolate a set of constraints that guarantees preservation of typability for security types, and investigate the impact of standard compiler optimizations on security types.
- Integrity: it is of practical interest, and we believe straightforward, to adapt our results to integrity. Indeed, weak forms of integrity guarantee that high variables may not be modified by a low writer, and are dual to confidentiality.

References

1. T. Ball. What's in a region? Or computing control dependence regions in near-linear time for reducible control flow. *ACM Letters on Programming Languages and Systems*, 2(1–4):1–16, March–December 1993.
2. A. Banerjee and D. A. Naumann. Secure Information Flow and Pointer Confinement in a Java-like Language. In *Proceedings of CSFW'02*. IEEE Computer Society Press, 2002.
3. G. Barthe and B. Serpette. Partial evaluation and non-interference for object calculi. In A. Middeldorp and T. Sato, editors, *Proceedings of FLOPS'99*, volume 1722 of *Lecture Notes in Computer Science*, pages 53–67. Springer-Verlag, 1999.

[2] See `http://www.dcs.ed.ac.uk/home/mrg`

4. C. Bernardeschi and N. De Francesco. Combining Abstract Interpretation and Model Checking for analysing Security Properties of Java Bytecode. In A. Cortesi, editor, *Proceedings of VMCAI'02*, volume 2294 of *Lecture Notes in Computer Science*, pages 1–15, 2002.
5. P. Bieber, J. Cazin, V.Wiels, G. Zanon, P. Girard, and J.-L. Lanet. Checking Secure Interactions of Smart Card Applets: Extended version. *Journal of Computer Security*, 10:369–398, 2002.
6. A. Coglio. Simple Verification Technique for Complex Java Bytecode Subroutines. In *Proceedings of FTFJP'02*, 2002.
7. J. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of SOSP'82*, pages 11–22. IEEE Computer Society, 1982.
8. N. Heintze and J. Riecke. The SLam calculus: programming with secrecy and integrity. In *Proceedings of POPL'98*, pages 365–377. ACM Press, 1998.
9. K. Honda and N. Yoshida. A Uniform Type Structure for Secure Information Flow. In *Proceedings of POPL'02*, pages 81–92. ACM Press, 2002.
10. C. League, Z. Shao, and V. Trifonov. Precision in Practice: A Type-Preserving Java Compiler. In G. Hedin, editor, *Proceddings of CC'03*, volume 2622 of *Lecture Notes in Computer Science*, pages 106–120. Springer-Verlag, 2003.
11. X. Leroy. Java bytecode verification: an overview. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of CAV'01*, volume 2102 of *Lecture Notes in Computer Science*, pages 265–285. Springer-Verlag, 2001.
12. A.C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of POPL'99*, pages 228–241. ACM Press, 1999.
13. G. C. Necula. Proof-Carrying Code. In *Proceedings of POPL'97*, pages 106–119. ACM Press, 1997.
14. G. C. Necula and P. Lee. The Design and Implementation of a Certifying Compiler. In *Proceedings of PLDI'98*, pages 333–344, 1998.
15. F. Pottier and V. Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, January 2003.
16. A. Sabelfeld and A. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Comunications*, 21:5–19, January 2003.
17. V. Simonet. Fine-grained Information Flow Analysis for a Lambda-Calculus with Sum Types. In *Proceedings of CSFW'02*, pages 223–237, 2002.
18. D. Volpano and G. Smith. A Type-Based Approach to Program Security. In M. Bidoit and M. Dauchet, editors, *Proceedings ofTAPSOFT'97*, volume 1214 of *Lecture Notes in Computer Science*, pages 607–621. Springer-Verlag, 1997.
19. D. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *Proceedings of CSFW'97*, pages 156–168. IEEE Press, 1997.
20. D. Volpano, G. Smith, and C. Irvine. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, pages 167–187, December 1996.
21. S. Zdancewic and A. Myers. Secure information flow and CPS. In D. Sands, editor, *Proceedings of ESOP'01*, volume 2028 of *Lecture Notes in Computer Science*, pages 46–61. Springer-Verlag, 2001.

History-Dependent Scheduling for Cryptographic Processes

Vincent Vanackère

Laboratoire d'Informatique Fondamentale de Marseille
Université de Provence,
39 rue Joliot-Curie, 13453 Marseille, France
vanackere@cmi.univ-mrs.fr

Abstract. This paper presents *history-dependent scheduling*, a new technique for reducing the search space in the verification of cryptographic protocols. This technique allows the detection of some "non-minimal" interleavings during a depth-first search, and was implemented in TRUST, our cryptographic protocol verifier. We give some experimental results showing that our method can greatly increase the efficiency of the verification procedure.

1 Introduction

In recent years, several symbolic reduction systems have been introduced, that aim at the analysis of security protocols. Some examples of this approach include [7,1,4]. These symbolic methods have an advantage over more traditional model-checking approaches in that they allow for the exploration and verification of an otherwise infinite-branching system, making it possible to perform an exact analysis for systems with a finite number of cryptographic processes.

However, the same problem remains as in most other model-checking techniques: as the number of parallel processes goes up, the number of possible interleavings makes the verification task harder – if not impossible in practice – because of the state-space explosion problem.

In this paper we present *history-dependent scheduling*, a new reduction technique that has been developed to be used in TRUST [10,11], our cryptographic protocol verifier. This technique allows, in a depth-first search setting, the detection of some redundant – "non-minimal" – interleavings, and is also well-suited to symbolic transition systems, where few transitions actually commute and the usual reduction methods such as those of [6] do not apply.

The paper is organised as follows. After the presentation of our formal model, we will give an intuitive overview of our reduction procedure. This procedure will then be formally described and proved in the next two sections. The paper will end with some experimental results and concluding remarks.

B. Steffen and G. Levi (Eds.): VMCAI 2004, LNCS 2937, pp. 16–29, 2004.

2 Model

Our full model is presented in details in [2], and we will therefore only give a short version here.

We work under the usual Dolev-Yao model [5], where the network is under full control of an adversary that can analyse all messages exchanged and synthetize new ones. In our setting, messages can be viewed as terms in a free algebra – we work under the perfect encryption assumption – and we distinguish between basic names (agent's names, nonces, keys, ...) and composed messages (pairs $< \_, \_ >$ and encrypted terms $E(\_, \_)$), with the restriction that only basic names may be used as encryption keys. The set of names is denoted by $\mathcal{N}$ and the full set of messages by $\mathcal{M}$.

2.1 The Intruder's Knowledge

Given a set of terms T, we will denote the set of terms that the intruder may derive by $\mu(T)$. We assume a (computable) relation $\mathcal{D} \subseteq \mathcal{N} \times \mathcal{N}$ with the following interpretation:

$$(C, C') \in \mathcal{D} \quad \text{iff} \quad \text{messages encrypted with } C \text{ can be decrypted with } C'.$$

We define $Inv(C) = \{C' \mid (C, C') \in \mathcal{D}\}$. Further hypotheses on the properties of $\mathcal{D}$ allow to model hashing, symmetric, and public keys. In particular: (i) for a *hashing* key C, $Inv(C) = \emptyset$, (ii) for a *symmetric* key C, $Inv(C) = \{C\}$, and (iii) for a *public* key C there is another key C' such that $Inv(C) = \{C'\}$ and $Inv(C') = \{C\}$.

Given a set of terms T we define the S (synthesis) and A (analysis) operators as follows:

- $S(T)$ is the least set that contains T and such that:

$$\begin{array}{ll} t_1, t_2 \in S(T) & \Rightarrow \ \langle t_1, t_2 \rangle \in S(T) \\ t_1 \in S(T), t_2 \in T \cap \mathcal{N} & \Rightarrow \ E(t_1, t_2) \in S(T) \ . \end{array}$$

- $A(T)$ is the least set that contains T and such that:

$$\begin{array}{ll} \langle t_1, t_2 \rangle \in A(T) & \Rightarrow \ t_i \in A(T), \ i = 1, 2 \\ E(t_1, t_2) \in A(T), A(T) \cap Inv(t_2) \neq \emptyset & \Rightarrow \ t_1 \in A(T) \ . \end{array}$$

As an example, if $T = \left\{E(\langle A, B \rangle, K), K^{-1}\right\}$ then $A(T) = T \cup \{A, B, \langle A, B \rangle\}$ and we have $E(A, K^{-1}) \in S(A(T))$.

With the above definitions, the knowledge that the intruder may derive from T is $\mu(T) = S(A(T))$. It should be noticed that the knowledge $\mu(T)$ is infinite whenever T is non-empty, and that it increases monotonically with T.

2.2 Processes and Configurations: Semantics

In our framework, a protocol is modelled as a finite number of processes interacting with an environment. Our process syntax includes the parallel composition – commutative and associative – of two processes, and thus we define a configuration k as a couple (P, T) where P is a process and T is an environment; the environment is a set of terms, composed of the initial knowledge augmented with all the messages emitted by the participants of the protocol so far. In the following, the adversary knowledge in a configuration $k \equiv (P, T)$ will be denoted by $\mu(k) = \mu(T)$.

Figure 1 gives the semantic rules as a reduction system on configurations. Informally, a process can either:

- (!) Write a message: the term is added to the environment knowledge.
- (?) Read some message from the environment: this can be any message the adversary is able to build from its current knowledge.
- (d) Decrypt some (encrypted) term with a corresponding inverse key.
- (pl) Perform some unpairing (the symmetric rule (pr) is not written).
- $(\mathsf{m}_1, \mathsf{m}_2)$ Test for the equality/inequality of two messages.
- (a) Check if some assertion φ holds in the current configuration.

$$
\begin{array}{lll}
(!) & (!t.P \mid P', T) & \to (P \mid P', T \cup \{t\}) \text{ if } t \in \mathcal{M} \\
(?) & (?x.P \mid P', T) & \to ([t/x]P \mid P', T) \text{ if } t \in \mu(T) \\
(\mathsf{d}) & (x \leftarrow \mathsf{dec}(E(t, C), C').P \mid P', T) & \to ([t/x]P \mid P', T) \text{ if } C' \in Inv(C), t \in \mathcal{M} \\
(\mathsf{pl}) & (x \leftarrow \mathsf{proj}_l(\langle t, t' \rangle).P \mid P', T) & \to ([t/x]P \mid P', T) \text{ if } t, t' \in \mathcal{M} \\
(\mathsf{a}) & (\mathsf{assert}(\varphi).P \mid P', T) & \to \begin{cases} (P \mid P', T) & \\ \mathsf{err} & \text{if } \not\models_T \varphi \end{cases} \\
(\mathsf{m}_1) & ([t = t]P_1, P_2 \mid P', T) & \to (P_1 \mid P', T) \text{ if } t \in \mathcal{M} \\
(\mathsf{m}_2) & ([t = t']P_1, P_2 \mid P', T) & \to (P_2 \mid P', T) \text{ if } t \neq t', t, t' \in \mathcal{M}
\end{array}
$$

Fig. 1. Reduction on configurations

Missing from the figure is the terminated process, denoted by 0, as well as the syntax of the assertion language, that will be presented in the next section. err denotes a special configuration that can only be reached from a false assertion.

In our model, *a correct protocol is a protocol that cannot reach the* err *configuration* – or, put in other words, a protocol such that all assertions reachable from the initial configuration of the system hold.

This reachability problem was shown to be NP-complete [2,9].

2.3 The Assertion Language

The full assertion language we consider is the following:

$$\varphi ::= \mathit{true} \mid \mathit{false} \mid t = t' \mid t \neq t' \mid \mathsf{known}(t) \mid \mathsf{secret}(t) \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2$$

This is equivalent to saying that we consider arbitrary boolean combinations of atomic formulas checking the equality of two messages ($t = t'$) and the secrecy of a message ($\mathsf{secret}(t)$) with respect to the current knowledge of the adversary. As shown in [2], this language allows to easily express authentication properties such as aliveness and agreement [8]. An actual example of a specification using this assertion language is given in Appendix A.

The valuation of an assertion formula in an environment T is the "intuitive" one: for example, we have $\models_T \mathsf{secret}(t)$ iff $t \notin \mu(T)$. The only property on assertions that is used in this paper is the fact that *the truth value of an assertion in an environment T depends only on the knowledge $\mu(T)$.*

2.4 Symbolic Reduction and Challenges for Practical Verification

The main difficulty in the verification task is the fact that the input rule is infinitely branching as soon as the environment is not empty. In [1,2] it was shown that it is possible to solve this problem by using a symbolic reduction system that stores the constraints in a symbolic shape during the execution. As an example, the input rule $(?x.P, T) \rightarrow ([t/x]P, T), t \in \mu(T)$ becomes $(?x.P, T, E) \rightarrow (P, T, (E\,;\, x : T))$. The complete description of the symbolic reduction system can be found in [2]. The main property we rely on is the fact that the symbolic reduction system is in lockstep with the ground one and provides a – sound and complete – decision procedure for processes specified using the full assertion language described in Sect. 2.3.

Although the symbolic reduction system is satisfying from a theoretical point of view, an inherent limitation is that it does not handle iterated processes (as the general case for iterated processes is undecidable). Thus, in order to verify a protocol against replay attacks and/or parallel sessions attacks, it is important to handle cases where there is a finite – even if small – number of participants playing each role.

Of course, as the number of parallel threads goes up, the number of possible interleavings make the verification task harder – if not impossible in practice – because we face the classical state explosion problem. Usual model-checking techniques to handle this problem involve the use of partial-order reduction [6], but unfortunately the symbolic reduction system doesn't lend itself very well to this approach, as different sequences of symbolic transitions will usually not lead to the same symbolic state.

We will tackle this problem by using a different approach: the basic idea is that by monitoring the inputs/outputs of the processes, we will be able to explore only some "complete" set of "interesting" interleavings. Due to the relative complexity of our reduction procedure, we will start by giving an intuitive overview in the next section.

3 A Scheduler for Cryptographic Processes

This section provides the intuition behind our reduction procedure. First, we recall the most important facts about our system:

- We consider a parallel composition of n processes.
- Processes can only interact through the environment.
- Each reduction step occurs only on one process, and its effect only depends on the environment knowledge.
- The environment knowledge can only increase along the reduction.

We will tackle our verification problem by using the following point of view:

- To each process of the parallel composition is associated a unique "nice level" (in the following, we will, without loss of generality, assume that the nice level of a process is its number in the parallel composition).
- The verification procedure is performed by a scheduler which is in charge, at each step of the reduction, of choosing the next process to be reduced, based on information on the past of the reduction. The scheduler may also choose to stop the exploration of the current branch.

This scheduler is similar to the one in a usual operating system, in the sense that it makes use of information on the past to make his decisions; the analogy stops here because our verification scheduler does not aim at achieving fairness or latency, but only at completeness. Moreover, it is also non-deterministic and has the possibility to sequentially explore several branches starting from the same configuration. Obviously, if we take as a scheduler the one exploring all possible branches at each step of the reduction, our description is just another name for a depth-first search (but, of course, we will try to be more clever...).

Our reduction procedure can now be (informally) illustrated as a scheduler that does the following:

Eager reduction: If the scheduler has selected some process for reduction, then this process may as well remain selected until it modifies the environment knowledge, else the reduction done on this process would not be able to affect the possible reductions of the other processes. From now on, we will call "eager step of reduction" any succession of reductions on the same process where the last step strictly increases the environment knowledge, and we will assume that the scheduler only performs eager steps of reduction.

Minimal traces: Whenever the scheduler stops reducing some process and then, later, decides to resume the reduction of this process, it will expect that the new eager step of reduction:

1. could not have happened at the time the reduction on this process was stopped ("no late work" clause);
2. could not have happened before the reduction of another process with a higher nice level ("priority" clause).

If any of those 2 conditions does not hold, the scheduler will simply stop the exploration of the current reduction.

We will now focus at formally defining and proving this reduction procedure. Section 4 will focus on the *eager reduction* procedure (this notion was already briefly described in [10]). Then, in Sect. 5 we will introduce a notion of *minimal traces* and give some criterions allowing the detection of non-minimal traces.

4 Eager Reduction

This section focuses on the description of the eager reduction procedure, that will be proven correct and complete w.r.t. the original reduction system.

In the following, we study the reduction of a configuration $(P_1 \mid \ldots \mid P_m, T)$, denoted by $(\Pi P_i, T)$. We will not allow the rewrite of $P \mid Q$ as $Q \mid P$, and therefore we can define the relation $\rightarrow_x$ as a reduction on the x-th process of the parallel composition. The variables $k, k', \ldots$ will be used for configurations: we recall that if $k \equiv (\Pi P_i, T)$, then $\mu(k)$ is defined as the environment knowledge in that configuration ($\mu(k) = \mu(T) = S(A(T))$), and if $k = \mathsf{err}$ we set $\mu(\mathsf{err}) = \emptyset$.

Reduction steps that do not modify the environment knowledge will play an important role in the following, and will be denoted as "silent":

Definition 4.1 (Silent reduction).
We will note $k \xrightarrow{\tau} k'$ *iff* $k \rightarrow k'$ *and* $\mu(k') = \mu(k)$.

The following commutation lemma for silent reductions, whose proof can be found in Appendix B, plays a crucial role in our reduction procedure:

Lemma 4.1 (Commutation lemma).
If $i \neq j$,
(1) $k \xrightarrow{\tau}_i \cdot \rightarrow_j \mathsf{err} \Rightarrow k \rightarrow_j \mathsf{err}$
(2) $k \xrightarrow{\tau}_i \cdot \rightarrow_j k' \neq \mathsf{err} \Rightarrow k \rightarrow_j \cdot \xrightarrow{\tau}_i k'$

A corollary of this lemma is the fact that if we consider a sequence of silent reductions $k \xrightarrow{\tau} {}^* k'$, then we can choose – in the path from k to k' – any order to reduce the processes of the parallel composition.

We will now annotate sequences of reductions to include the order in which the different processes modify the environment knowledge. In the following, we will write $\twoheadrightarrow$ as a shortcut for $\rightarrow^*$. In a similar way, we will write $\twoheadrightarrow_x \equiv (\rightarrow_x)^*$ for a sequence of reductions occurring on the process x.

Definition 4.2. *We write:*

$$
\begin{array}{lll}
(1)\ k \overset{\tau}{\twoheadrightarrow} k' & \textit{iff} & k \twoheadrightarrow k' \textit{ and } \mu(k') = \mu(k) \\
(2)\ k \overset{x}{\twoheadrightarrow} k' & \textit{iff} & k \overset{\tau}{\twoheadrightarrow} \cdot \rightarrow_x \cdot \overset{\tau}{\twoheadrightarrow} k' \textit{ and } \mu(k') \neq \mu(k) \\
(3)\ k \overset{x_1,\ldots,x_n}{\twoheadrightarrow} k' & \textit{iff} & k \overset{x_1}{\twoheadrightarrow} \cdot \overset{x_2}{\twoheadrightarrow} \cdots \overset{x_n}{\twoheadrightarrow} k'
\end{array}
$$

The relation $\overset{\tau}{\twoheadrightarrow}$ stands for any sequence of silent reductions ; $\overset{x}{\twoheadrightarrow}$ means that the environment knowledge is modified (increased) exactly once, by the process x, during the reduction. Notation $\overset{x_1,\ldots,x_n}{\twoheadrightarrow}$ is only syntactic sugar: intuitively, the sequence $\{x_1, \ldots, x_n\}$ represents the order in which the processes bring new information to the environment.

Remark 4.1 (An output can hide behind another ...).

If $k \twoheadrightarrow k'$ then there exists a sequence $\{x_1, \ldots, x_n\}$ such that $k \overset{x_1,\ldots,x_n}{\twoheadrightarrow} k'$. It should be noted that the set of sequences s satisfying $k \overset{s}{\twoheadrightarrow} k'$ is not related

to Mazurkiewicz traces[1]; as an example, if $k = (!A.P_1 \mid !\langle A, B\rangle.P_2,\ \{B\})$ and $k' = (P_1 \mid P_2,\ \{A, B, \langle A, B\rangle\})$, both the sequences $\{1\}$ and $\{2\}$ satisfy $k \overset{s}{\twoheadrightarrow} k'$ (after the output from either one of the processes, an output from the other one will not bring any new information to the environment).

Definition 4.3 (Eager reduction).
We define $\hookrightarrow_x$, a step of eager reduction on the process x, as:

$$k \hookrightarrow_x k' \text{ iff } k \overset{\tau}{\twoheadrightarrow}_x \cdot \rightarrow_x k' \text{ and } \mu(k') \neq \mu(k)\ .$$

The eager reduction relation $\hookrightarrow$ is then defined by $\hookrightarrow = \bigcup_x \hookrightarrow_x$.

By extension, we will write $k \hookrightarrow_{x_1,\dots,x_n} k'$ whenever $k \hookrightarrow_{x_1} \cdot \hookrightarrow_{x_2} \cdots \hookrightarrow_{x_n} k'$.

Thus, during a step of eager reduction, we reduce only some process x of the configuration until it increases the environment knowledge or reaches error: this formal definition is the one corresponding to rule (1) of our scheduler.

By using Lemma 4.1, we can establish the following theorem on eager reduction (the proof can be found in Appendix C):

Theorem 4.1.

1. $k \overset{x_1,\dots,x_n}{\twoheadrightarrow} k' \neq \mathsf{err} \Rightarrow k \hookrightarrow_{x_1,\dots,x_n} \cdot \overset{\tau}{\twoheadrightarrow} k'$
2. $k \overset{x_1,\dots,x_n}{\twoheadrightarrow} \mathsf{err} \Rightarrow k \hookrightarrow_{x_1,\dots,x_n} \mathsf{err}$

We can now state the soundness and completeness of the eager reduction:

Theorem 4.2.

$$k \rightarrow^* \mathsf{err} \text{ iff } k \hookrightarrow^* \mathsf{err}$$

Proof. Completeness follows from Theorem 4.1(2): if $k \twoheadrightarrow \mathsf{err}$, then there exists $\{x_1, \dots, x_n\}$ such that $k \overset{x_1,\dots,x_n}{\twoheadrightarrow} \mathsf{err}$, and thus $k \hookrightarrow_{x_1,\dots,x_n} \mathsf{err}$, which means $k \hookrightarrow^* \mathsf{err}$. Soundness comes trivially from $\hookrightarrow \subseteq \rightarrow^*$. □

5 Characteristics and Minimal Traces

In this section we will show how it is possible to avoid the full exploration of all eager traces by exploiting some information on the past of a trace. It is namely possible, by monitoring the values taken in input, to detect, at the end of its execution, that some step of reduction could have occurred earlier in the trace; by using a suitable order on traces (Definition 5.2), we will show that we can cut this branch of the search space whenever we detect that the current trace cannot be a minimal one.

In all this section, we consider an initial configuration k such that $k \rightarrow^* \mathsf{err}$ and we will show the existence of a reduction sequence from k to err verifying some particular properties.

[1] We recall that two words/traces are Mazurkiewicz-equivalent iff they can be obtained by permutations of independent letters/transitions.

5.1 Definitions

Definition 5.1 (Traces and characteristics).
An eager trace *from k to k' is a sequence $k \hookrightarrow k_1 \hookrightarrow \cdots \hookrightarrow k'$.*
We will denote $k \hookrightarrow_{p^m} k'$ whenever $k \hookrightarrow_p \cdots \hookrightarrow_p k'$ with m eager steps.
Any eager trace can be uniquely written as:

$$k \hookrightarrow_{p_1{}^{m_1},\ldots,p_n{}^{m_n}} k' \text{ with } \forall i \;\; p_i \neq p_{i+1} \text{ and } m_i > 0 \;\; .$$

The characteristic of this trace is then defined by the tuple $(p_1{}^{m_1}, \ldots, p_n{}^{m_n})$.

In the following, we will often write "trace" as a shortcut for "eager trace". We now introduce a way to compare the traces characteristics:

Definition 5.2 (Partial order on characteristics).
The relation $\prec$ is defined as:

$$(p_1{}^{m_1}, \ldots, p_n{}^{m_n}) \prec (p'_1{}^{m'_1}, \ldots, p'_{n'}{}^{m'_{n'}})$$

$$\text{iff} \quad \exists i \in \{1, \ldots, \min(n, n')\} \begin{cases} \forall j < i \;\; (p_j, m_j) = (p'_j, m'_j) \\ p_i < p'_i \vee (p_i = p'_i \wedge m_i > m'_i) \end{cases}$$

Example 5.1. If $a < b < c$, then $(a^3, b^1, c^2) \prec (a^3, c^2) \prec (a^3, c^1, b^7)$.

The introduction of this particular relation could appear counter-intuitive, due to condition $m_i > m'_i \ldots$ The informal explanation is the following: the purpose of this order is to set as minimal the traces for which *stopping the reduction on some process implies that the next reduction on this process must necessarily depend on what happened "in-between"*. Then, if $k \hookrightarrow_a \cdot \hookrightarrow_a \cdot \hookrightarrow_b k'$ and $k \hookrightarrow_a \cdot \hookrightarrow_b \cdot \hookrightarrow_a k'$, the first sequence will have a smaller characteristic than the second one (and should ideally be favoured by our scheduler).

It should be noted that our relation makes use of an arbitrary order on the processes – their number – that corresponds in fact to the "nice levels" of our informal scheduler description.

Lemma 5.1. *$\prec$ is a partial order.*

Remark 5.1. In fact, two characteristics are always comparable unless one is the prefix of the other.

The following lemma will allow us to establish the existence of minimal traces leading to error:

Lemma 5.2. *The set of all characteristics associated to some non-empty set of traces from the same initial configuration admits some minimal elements.*

Proof. Although $\prec$ is not well founded this property simply holds because there is only a finite number of possible characteristics for all the traces starting from a given initial configuration k (namely if W is the total number of output instructions contained in k, any eager trace will be at most of length $W + 1$). □

Definition 5.3 (Minimal traces).
We consider the set of all characteristics associated to all traces from k to err*: this set admits minimal elements, and any trace from k to* err *whose characteristic is minimal will be called minimal.*

As a consequence, finding criteria for non-minimal traces will allow us to avoid the full exploration of all traces by the scheduler.

5.2 Criteria for Non-minimal Traces

In this section, we will develop a characterisation of some non-minimal traces, based on the observation of the values taken by the input variables of the processes. This will require a new notation:

Definition 5.4. *Let X be a set of terms. We denote:*

$$(?x.P, T) \overset{?X}{\to} ([t/x]P, T) \quad \textit{iff} \quad t \in S(A(T)) \cap X$$

By extension, we will also denote $k \overset{?X}{\hookrightarrow}_p k'$ when all the input values in the reduction from k to k' are included in the set X.

The following – crucial – lemma is proven in Appendix D:

Lemma 5.3 (Eager commutation).

If $i \neq j$, $X \subseteq \mu(k)$ and $k \hookrightarrow_i \cdot \overset{?X}{\hookrightarrow}_j \cdot \hookrightarrow^+ \mathsf{err}$, then $k \overset{?X}{\hookrightarrow}_j \cdot \hookrightarrow^+ \mathsf{err}$.

We can now state our fundamental theorem, that gives two sufficient conditions for the non-minimality of a trace:

Theorem 5.1 (Non-minimality). *Any trace $k_1 \hookrightarrow_{p_1{}^{m_1}} \cdots \hookrightarrow_{p_n{}^{m_n}} \mathsf{err}$ containing a sub-trace of one of the following forms is not minimal (we denote $X_i \equiv \mu(k_i)$):*

$$\begin{array}{ll}
(1) & k_i \hookrightarrow_{p_i{}^{m_i}} k_{i+1} \cdots \hookrightarrow_{p_j{}^{m_j}} \cdot \overset{?X_{i+1}}{\hookrightarrow}_p k' \neq \mathsf{err} \\
& p = p_i,\ p \notin \{p_{i+1}, \dots, p_j\} \\
(2) & k_i \hookrightarrow_{p_i{}^{m_i}} \cdots \hookrightarrow_{p_j{}^{m_j}} \cdot \overset{?X_i}{\hookrightarrow}_p k' \neq \mathsf{err} \\
& p < p_i,\ p \notin \{p_i, \dots, p_j\}
\end{array}$$

Proof.

(1) By iterating Lemma 5.3, we show $k_i \hookrightarrow_{p_i{}^{m_i+1}} \cdot \hookrightarrow^+ \mathsf{err}$. Therefore there exists some trace of characteristic $(p_1{}^{m_1}, \dots, p_i{}^{m_i+q}, \dots)$ leading to error, with $q \geq 1$. However, by definition of $\prec$ we have $(p_1{}^{m_1}, \dots, p_i{}^{m_i+q}, \dots) \prec (p_1{}^{m_1}, \dots, p_i{}^{m_i}, \dots)$, thus the non-minimality of our initial trace.

(2) By iterating Lemma 5.3, we get $k_i \hookrightarrow_p \cdot \hookrightarrow^+ \mathsf{err}$. If $i > 1$ and $p = p_{i-1}$, we are back to the previous case, else there exists some trace with characteristic $(p_1{}^{m_1}, \dots, p_{i-1}{}^{m_{i-1}}, p^q, \dots)$ leading to error, with $q \geq 1$. As $p < p_i$, we have $(p_1{}^{m_1}, \dots, p^q, \dots) \prec (p_1{}^{m_1}, \dots, p_i{}^{m_i}, \dots)$, and our initial trace is not minimal.

5.3 Configurations with History

Theorem 5.1 shows that by having the relevant information on the past of the current trace, our scheduler could easily detect – and discard – non-minimal traces. We will therefore enrich configurations with a third component, the *history*, which will be a list of triple; for each reduction sequence on one process, we will indeed record the process number, the current environment just at the beginning of the reduction on this process, and the list of all the values that were read during the sequence. Formally, we write: $\mathcal{H} = (p_0, T_0, X_0) : \cdots : (p_n, T_n, X_n)$, where each p_i is a process number, T_i an environment and X_i a set of terms.

We define a function append_H for updating a history as follows:

Definition 5.5. *If* $\mathcal{H} = (p_0, T_0, X_0) : \cdots : (p_n, T_n, X_n)$*, then:*

- *if* $p = p_n$, $\mathsf{append}_H(\mathcal{H}, p, T, X) = (p_0, T_0, X_0) : \cdots : (p_n, T_n, X_n \cup X)$
- *if* $p \neq p_n$, $\mathsf{append}_H(\mathcal{H}, p, T, X) = (p_0, T_0, X_0) : \cdots : (p_n, T_n, X_n) : (p, T, X)$.

Reduction rules are then modified to integrate a history:

- the input rule becomes:

$$(?x.P \mid P', T, \mathcal{H}) \rightarrow_i ([t/x]P \mid P', T, \mathcal{H}') \text{ if } \begin{cases} t \in S(A(T)) \\ \mathcal{H}' = \mathsf{append}_H(\mathcal{H}, i, T, \{t\}) \end{cases}$$

- for all the other rules, if $(P, T) \rightarrow_i (P', T')$ then:

$$(P, T, \mathcal{H}) \rightarrow_i (P', T', \mathcal{H}') \text{ if } \mathcal{H}' = \mathsf{append}_H(\mathcal{H}, i, T, \emptyset) \ .$$

Put in other words, each time the scheduler will choose a new process for reduction, it will start a new section of the history and record the current environment; this section will then be updated with all the values that are read by this process during its execution/reduction.

We can now state our final theorem, giving the two conditions that will be checked, after each eager step, by the scheduler in order to discard non-minimal traces:

Theorem 5.2 (Non-minimal history). *We consider a trace* $k \hookrightarrow^+ k' \neq \mathsf{err}$ *such that the history* $\mathcal{H}(k') = (p_0, T_0, X_0) : \cdots : (p_n, T_n, X_n)$ *satisfies one of the following properties:*

$$(1)\ \exists i < n. \begin{cases} p_i = p_n \\ p_n \notin \{p_{i+1}, \ldots, p_{n-1}\} \\ X_n \subseteq \mu(T_{i+1}) \end{cases}$$

$$(2)\ \exists i < n. \begin{cases} p_i > p_n \\ p_n \notin \{p_{i+1}, \ldots, p_{n-1}\} \\ X_n \subseteq \mu(T_i) \end{cases}$$

Then this trace cannot be the prefix of a minimal one.

Proof. $\mathcal{H}(k')$ verifies either (1) or (2) and therefore the eager reduction to k' will verify the corresponding condition from Theorem 5.1. As a consequence, any trace having the one to k' as a prefix cannot be a minimal one.

6 Experimental Results

We have implemented our history-based reduction procedure in our verifier, TRUST. Figure 2 gives some times for the full analysis of some typical protocols: the measures were taken on an Athlon XP 1800, and all times are in seconds (the time spent by the verifier is roughly proportional to the number of basic reduction steps that are done). For each protocol, we detail the number of roles involved and give the times to do a full search depending on the number of parallel – interleaved – sessions[2]. T_0 is the time when the usual reduction method is applied (*i.e.* the scheduler reduces each process until it emits an output); T_{Eager} is the time for the eager reduction procedure alone; T_{Min} is the time without eager reduction, but using the history to detect non-minimal traces; then $T_{\text{Eager+min}}$ is the time for the full reduction procedure presented in this paper. As can be seen in the figure, our reduction method reveals itself quite effective in practice: sometimes a reduction factor of 60 is gained, and we did not encounter any case where the added checks (for non-minimality) made history-dependent scheduling slower.

Protocol	# roles	# sessions	T_0	T_{Eager}	T_{Min}	$T_{\text{Eager+Min}}$
Needham-Schroeder-Lowe	2	3	50	6.80	1.18	0.59
	2	4	?	2034	99	41
Needham-Schroeder-Lowe[3]	3	2	277	2.38	1.04	0.16
	3	3	?	963	163	8.46
Otway-Rees	3	2	0.75	0.32	0.28	0.14
	3	3	5879	722	497	98
Carlsen	3	2	6.15	4.79	1.29	0.91
	3	3	?	?	?	2272
Kerberos v5	4	2	5.40	4.11	2.31	1.93

Fig. 2. Times for the analysis of various protocols (in seconds).

7 Conclusion

In this paper we have presented history-dependent scheduling, a new reduction method for cryptographic protocols that is based on the monitoring of the inputs/outputs performed by the processes during the reduction. This method relies on the two following techniques:

[2] A question mark instead of the time means that we were not "patient" enough to wait for the end of the verification procedure.

[3] This is the seven-message version of the protocol, making use of 3 key servers in parallel.

- Eager reduction, that was already presented in [10], is a natural big-step semantics based on the outputs of the different processes.
- Detection of non-minimal traces, through the use of a history of the current reduction sequence, allows to stop the exploration of some traces and is based on very simple criteria on the inputs of the processes.

As far as related work is concerned, our technique seems to share some common grounds with the one developed independently in [3], in that both methods somehow aim to reduce the search space by verifying/maintaining additional constraints on the input values; however, differences between the formalisms make the comparison a non-trivial task, that we have to leave as a future work.

It should be noted that we have only defined and proved here our method on the ground reduction system; this method can be adapted in a straightforward manner to the symbolic system, and the same non-minimality conditions are then checked at the symbolic level (although we should mention that the proof of completeness for the symbolic case becomes more complex, due to some slight differences between the symbolic eager reduction and the ground one).

As practical experiments show, history-dependent scheduling can be quite effective in practice – and in all our tests never induces any slowdown. We expect that the ideas behind this method are general enough to be applied to other verification systems, and also stress the fact that this technique is, by its nature, especially well suited for verification in a depth-first setting.

References

1. R. Amadio and D. Lugiez. On the reachability problem in cryptographic protocols. In *Proc. CONCUR00, Springer LNCS 1877*, 2000. Also RR-INRIA 3915.
2. R. Amadio, D. Lugiez and V. Vanackère. On the symbolic reduction of processes with cryptographic functions. RR-INRIA 4147, March 2001. Revised version in *Theoretical Computer Science*, 290(1);695-740, 2003.
3. D. Basin, S. Mödersheim and L. Viganò. Constraint Differentiation: A New Reduction Technique for Constraint-Based Analysis of Security Protocols. Technical Report 405, Dep. of Computer Science, ETH Zurich, 2003.
4. M. Boreale. Symbolic Trace Analysis of Cryptographic Protocols. In *Proc. ICALP01, Springer LNCS, Berlin*, 2001.
5. D. Dolev and A. Yao. On the security of public key protocols. *IEEE Trans. on Information Theory*, 29(2):198–208, 1983.
6. P. Godefroid. Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. In *Springer LNCS 1032*, 1996.
7. A. Huima. Efficient infinite-state analysis of security protocols. In *Proc. Formal methods and security protocols, FLOC Workshop, Trento*, 1999.
8. G. Lowe. A hierarchy of authentication specifications. In *Proc. 10th IEEE Computer Security Foundations Workshop*, 1997.
9. M. Rusinowitch and M. Turuani. Protocol insecurity with finite number of sessions is NP-complete. RR INRIA 4134, March 2001.
10. V. Vanackère. The TRUST protocol analyser, automatic and efficient verification of cryptographic protocols. In *VERIFY'02 Workshop, Copenhagen*, 2002.
11. http://www.cmi.univ-mrs.fr/∼vvanacke/trust/

A Specifying Security Properties through Assertions

We take as an example the following 3 message version of the Needham-Schroeder Public Key protocol:

$$A \to B : \{na, A\}_{Pub(B)}$$
$$B \to A : \{na, nb\}_{Pub(A)}$$
$$A \to B : \{nb\}_{Pub(B)}$$

In our framework, the protocol can be modeled as follows:

$$\begin{aligned} Init(myid, resp) : \ & \text{fresh } na. \\ & !E(\langle na, myid\rangle, Pub(resp)). \\ & ?e.\ \langle na', nb\rangle \leftarrow \mathsf{dec}(e, Priv(myid)).\ [na' = na]. \\ & !E(nb, Pub(resp)). \\ & \mathsf{assert}(\mathsf{secret}(nb) \wedge \mathsf{auth}(resp, myid, na, nb)).\ 0 \end{aligned}$$

$$\begin{aligned} Resp(myid, init) : \ & ?e.\ \langle na, a\rangle \leftarrow \mathsf{dec}(e, Priv(myid)).\ [a = init]. \\ & \text{fresh } nb. \\ & !_{\mathsf{auth}(myid,init,na,nb)}\ E(\langle na, nb\rangle, Pub(init)). \\ & ?e'.\ [e' = E(nb, Pub(myid))]. \\ & 0 \end{aligned}$$

The assertion "auth(msg)" is a shortcut for "$\mathsf{known}(E(msg, K_{\mathsf{auth}}))$", and instruction "$!_{\mathsf{auth}(msg)}t$" is some syntactic sugar for "$!\langle E(msg, K_{\mathsf{auth}}), t\rangle$". In this example, the initiator specifies that at the end of its run of the protocol the nonce nb must be secret, and expects an agreement with some responder on the nonces na and nb.

B Proof of Lemma 4.1

(1) We have $k \xrightarrow{\tau}_i k' \to_j \mathsf{err}$. The reduction on j is a false assertion, and as we have $\mu(k) = \mu(k')$ and as the truth value of an assertion only depends on the environment knowledge, the same assertion will also evaluate to false in the configuration k.

(2) We have $k_1 \xrightarrow{\mathsf{r}_1}_i k_2 \xrightarrow{\mathsf{r}_2}_j k_3$ ($k_1 = k$ and $k_3 = k'$). The proof is done by basic case analysis on the rules r_1 and r_2. . All rules but (?), (a) and (!) do not depend at all from the environment nor modify, it and thus the result holds whenever $\to_i \notin \{(\mathsf{a}), (?), (!)\}$ or $\to_j \notin \{(\mathsf{a}), (?), (!)\}$. Commutation when $\mathsf{r}_1 = (\mathsf{a})$ or $\mathsf{r}_2 = (\mathsf{a})$ is always possible, given the fact that a process may choose not to evaluate an assertion. There remains only 4 cases:

1. $k_1 \xrightarrow{?}_i k_2 \xrightarrow{?}_j k_3$
2. $k_1 \xrightarrow{!}_i k_2 \xrightarrow{!}_j k_3$
3. $k_1 \xrightarrow{?}_i k_2 \xrightarrow{!}_j k_3$
4. $k_1 \xrightarrow{!}_i k_2 \xrightarrow{?}_j k_3$

Cases (1), (2) and (3) are straightforward[4]. Case (4) is the interesting one (and the one where our eager reduction procedure will take advantage): namely we can perform the input rule first and then reach k_3 after an output from the process number i, due to the fact that the input rule only depends on the knowledge $\mu(k_2)$, which is the same as $\mu(k_1)$ since $k_1 \xrightarrow{\tau}_i k_2$. □

C Proof of Theorem 4.1

We simultaneously prove (1) and (2) by induction on n.

Case $(n = 1)$**.** We have to establish:

(1) $k \overset{x}{\twoheadrightarrow} k' \neq \mathsf{err} \Rightarrow k \hookrightarrow_x \cdot \overset{\tau}{\twoheadrightarrow} k'$

(2) $k \overset{x}{\twoheadrightarrow} \mathsf{err} \Rightarrow k \hookrightarrow_x \mathsf{err}$

Both properties are easily shown by iterating Lemma 4.1 in order to move all reductions on the process x at the beginning of the reduction sequence.

Case $(n > 1)$. By induction:

(1) $k \overset{x_1,\ldots,x_n}{\twoheadrightarrow} k'$ implies $k \overset{x_1,\ldots,x_{n-1}}{\twoheadrightarrow} k'' \overset{x_n}{\twoheadrightarrow} k'$ and by induction hypothesis we know that:

$$\exists k_{n-1}.\ k \hookrightarrow_{x_1,\ldots,x_{n-1}} k_{n-1} \overset{\tau}{\twoheadrightarrow} k'' \ .$$

Then $k_{n-1} \overset{\tau}{\twoheadrightarrow} k'' \overset{x_n}{\twoheadrightarrow} k'$, which means $k_{n-1} \overset{x_n}{\twoheadrightarrow} k'$, and by the result for case $n = 1$ we can deduce $\exists k_n.\ k_{n-1} \hookrightarrow_{x_n} k_n \overset{\tau}{\twoheadrightarrow} k'$.

(2) We have $k \overset{x_1,\ldots,x_{n-1}}{\twoheadrightarrow} k' \overset{x_n}{\twoheadrightarrow} \mathsf{err}$ and by induction hypothesis:

$$\exists k''.\ k \hookrightarrow_{x_1,\ldots,x_{n-1}} k'' \overset{\tau}{\twoheadrightarrow} k' \overset{x_n}{\twoheadrightarrow} \mathsf{err} \ .$$

Then $k'' \overset{x_n}{\twoheadrightarrow} \mathsf{err}$ and thus, by the result for case $n = 1$, $k'' \hookrightarrow_{x_n} \mathsf{err}$, which ends the proof. □

D Proof of Lemma 5.3

First, we will extend Lemma 4.1 with the two following properties $(i \neq j)$:

(1) If $k \rightarrow_i \cdot \overset{?X}{\rightarrow}_j k'$ and $X \subseteq \mu(k)$ then $k \overset{?X}{\rightarrow}_j \cdot \ \rightarrow_i k'$

(2) If $k \rightarrow_i k_1 \overset{!}{\rightarrow}_j k'$ and $\mu(k_1) \neq \mu(k')$ then $\exists k_2.\, k \overset{!}{\rightarrow}_j k_2 \rightarrow_i k' \wedge \mu(k) \neq \mu(k_2)$

By iterated application of (1), (2) and Lemma 4.1, we can now move the reduction steps on the process j at the beginning of the sequence and get:

$$\exists (k', k'').\ \ k \overset{?X}{\twoheadrightarrow}_j k' \overset{!}{\rightarrow}_j k'' \twoheadrightarrow_i \cdot \hookrightarrow^+ \mathsf{err} \wedge \mu(k') \neq \mu(k'') \ .$$

As $\mu(k') \neq \mu(k'')$, $k \overset{?X}{\twoheadrightarrow}_j k' \overset{!}{\rightarrow}_j k''$ implies $\exists q \geq 1.\ k \overset{?X}{\hookrightarrow}_{j^q} k''$. On the other hand, we know by the completeness of eager reduction that $k'' \hookrightarrow^+ \mathsf{err}$. Therefore $k \overset{?X}{\hookrightarrow}_{j^q} \cdot \hookrightarrow^+ \mathsf{err}$, which implies $k \overset{?X}{\hookrightarrow}_j \cdot \hookrightarrow^+ \mathsf{err}$. □

[4] It should be noted that (3) is "folklore" and used very broadly in the literature; we can summarize it as: "if we have in a parallel one process doing an input and another one an output, the output can always be done first without any loss".

Construction of a Semantic Model for a Typed Assembly Language*

Gang Tan, Andrew W. Appel, Kedar N. Swadi, and Dinghao Wu

Department of Computer Science
Princeton University
{gtan,appel,kswadi,dinghao}@cs.princeton.edu

Abstract. Typed Assembly Languages (TALs) can be used to validate the safety of assembly-language programs. However, typing rules are usually trusted as axioms. In this paper, we show how to build semantic models for typing judgments in TALs based on an induction technique, so that both the type-safety theorem and the typing rules can be proved as lemmas in a simple logic. We demonstrate this technique by giving a complete model to a sample TAL. This model allows a typing derivation to be interpreted as a machine-checkable safety proof at the machine level.

1 Overview

Safety properties of machine code are of growing concern in both industry and academia. If machine code is compiled from a safe source language, compiler verification can ensure the safety of the machine code. However, it is generally prohibitive to do verification on an industrial-strength compiler due to its size and complexity. In this paper, we do validation directly on machine code. Necula introduced Proof-Carrying Code (PCC) [1], where a low-level code producer supplies a safety proof along with the code to the consumer. He used types to specify loop invariants and limited the scope of the proof to type safety. Typed Assembly Language (TAL) [2] by Morrisett *et al.* refined PCC by proposing a full-fledged low-level type system and a type-preserving compiler that can automatically generate type annotations as well as the low-level code. Once an assembly-language program with type annotations is type checked, the code consumer is assured of type safety.

Take a typing rule from the Cornell TAL [2],

$$\frac{\Psi; \Delta; \Gamma \vdash r : \forall[\,].\Gamma' \qquad \Delta \vdash \Gamma \leq \Gamma'}{\Psi; \Delta; \Gamma \vdash \texttt{jmp}\ r}$$

which means that a "jump to register r" instruction type-checks if the value in r is a code pointer with precondition Γ', and the current type environment Γ is a subtype of Γ'. This rule is intuitively "correct" based on the semantics of the jump instruction. (In this paper we won't be concerned with Ψ, Δ, etc. of the Cornell TAL; we show this rule just to illustrate the complexity of that system's trusted axioms.)

* This research was supported in part by DARPA award F30602-99-1-0519 and by NSF grant CCR-0208601.

B. Steffen and G. Levi (Eds.): VMCAI 2004, LNCS 2937, pp. 30–43, 2004.

In the Cornell TAL system and its variant [3] , such typing rules are accepted as axioms. They are a part of the Trusted Computing Base (TCB). However, low-level type systems tend to be complex because of intricate machine semantics. Any misunderstanding of the semantics could lead to errors in the type system. League *et al.* [4] found an unsound proof rule in the SpecialJ [5] type system. In the process of refining our own TAL [6], we routinely find and fix bugs that can lead to unsoundness.

In systems that link trusted to untrusted machine code, errors in the TCB can be exploited by malicious code. A more foundational approach is to move the entire type system out of the TCB by proving the typing rules as lemmas, instead of trusting them as axioms; also by verifying the type-safety theorem: type checking of code implies the safety policy, or the slogan—well-typed programs do not go wrong.

1.1 A Foundational Approach

In the type-theory community, there is a long tradition of giving denotational semantics [7] to types and proving typing rules as lemmas. Appel and Felty [8] applied this idea to PCC and gave a semantic model to types and machine instructions in higher-order logic. In the following years, semantic models of types have been extended to include recursive types [9] and mutable references [10]. With these models, it is possible to reason locally about types and operations on values, but unfortunately no model has been provided to typing judgments such as $\Psi; \Delta; \Gamma \vdash \text{jmp } r$ and no method is provided to construct the safety proof for an entire program.

The main contribution of this paper is to use a good set of abstractions to give models to judgments so that both typing rules and the type-safety theorem can be mechanically verified in a theorem-proving system. Our approach is truly foundational in that only axioms in higher-order logic and the operational semantics of instructions need to be trusted; it has a minimal trusted base. Our approach can be viewed as a way to map a typing derivation to a machine-checkable safety proof at the machine level, because each application of a typing rule in the typing derivation can be seen as the use of a (proved) lemma.

1.2 Model of TALs

In this section, we give an informal overview of our model of a typed assembly language, particularly an induction technique to prove program safety. We will not refer to any specific TAL in this section.

A TAL program consists of two parts: code and type annotations. As an example, the following code snippet has a number of basic blocks; each one is given a label (like l_0) and is composed of a sequence of instructions. Each basic block has an associated type annotation (like ϕ_0) that is the basic block's precondition.

$$
\begin{array}{ll}
\phi_0 & l_0 : \texttt{add}\ 1, 1, 2 \\
 & \quad\ \texttt{jmp}\ l_3 \\
\phi_1 & l_1 : \texttt{ld}\ 2, 3 \\
 & \quad\ \ldots \\
\phi_m & l_m : \ldots
\end{array}
$$

Type annotations are generated by a type-preserving compiler from source language types. They serve as a specification (types as specifications). For instance, if ϕ_0 is $\{r_1 : \mathsf{int}\} \cap \{r_2 : \mathsf{box}\,(\mathsf{int})\}$, it expresses that register one is of type integer and register two is a pointer to an integer.

Since we need to show the type-safety theorem, the first ingredient is to define what the safety of code means. Following the standard practice in type theory, we define code is safe if it will not get stuck before it naturally stops. Note the trick here is to define the machine's operational semantics in such a way that the machine will get stuck if it violates the safety policy (see Section 2). To simplify the presentation, we treat only nonterminating programs that do not stop naturally.[1]

We first informally explore how to prove a TAL program is safe. We define "a label l in a state is safe for k steps" to mean that when the control of the state is at l, the state can run for k steps. The goal then is to prove that if entry condition ϕ_0 holds, label l_0 is safe for k steps for any natural number k. A natural thought is to show it by induction over k. The base case ($k = 0$) is trivial; the inductive case is to show label l_0 is safe for $k + 1$ steps given that it is safe for k steps. But at this moment we have no idea in which state the code will be after k steps, so we cannot prove that the state can go ahead one more step.

The solution is to do induction simultaneously over all labels, i.e. prove each label l_i is safe for $k + 1$ steps with respect to its precondition ϕ_i, assuming all labels l_j are safe for k steps with respect to ϕ_j. Let us take label l_0 in the example to see why this new induction works. Basic block l_0 has length two, and ends with a jump to label l_3, which has been assumed to be safe for k steps, provided that precondition ϕ_3 is met. Suppose by inspecting the two instructions in block l_0, we have concluded that l_0 is safe for two steps and after two steps ϕ_3 will be true. Combined with the assumed k-step safety of label l_3, label l_0 is safe for $k + 2$ steps, which implies that it is safe for $k + 1$ steps.

In this proof method, we still need to inspect each instruction in every block. For example, in block l_0, we check if the precondition ϕ_0 is enough to certify the safety of its two instructions and if ϕ_3 will be met after their execution. What we have described essentially is a proof method to combine small proofs about instructions into a proof about the safety of the whole program. In the rest of this section, we informally give models to typing judgments based on this technique. Before that, we first motivate what kind of typing judgments a TAL would usually have.

To type check a TAL program, the type system must have a wellformedness judgment for programs. Since a TAL program is composed of basic blocks, the wellformedness judgment for programs requires another judgment, a wellformedness judgment for basic blocks. Similarly, the type system should also have a wellformedness judgment for instructions.

The model of the wellformedness judgment for programs can be that all labels are safe with respect to their preconditions. In the following sections, we will develop abstractions so that this model can be written down in a succinct subtyping formula: $\Delta(C) \subset \Gamma$. The model of wellformedness of a basic block can be that this particular

[1] Every program can be transformed into this form by giving it a continuation at the beginning and letting the last instruction be a jump to this continuation. The continuation could return to the operating system, for example.

basic block is safe for $k+1$ steps assuming all the other basic blocks are safe for k steps. Based on the induction technique and this model, we can prove the typing rule that concludes the wellformedness of a program from the wellformedness of basic blocks. The model of wellformedness of instructions is similar to the one of basic blocks and we do not go into details at this stage.

In Section 2, we present the model of a RISC architecture (Sparc) and formally define the safety of code based on a particular safety policy (memory safety); Section 3 shows the syntax of a sample TAL; Section 4 shows an indexed model of types and its intuition. The material in these three sections has been described by other papers [11,6, 9] as part of the foundational PCC project; we briefly sketch them to set up a framework within which our proof method can be formally presented in section 5.

2 Safety Specification

Our machine model consists of a set of formulas in higher-order logic that specify the decoding and operational semantics of instructions. Our safety policy specifies which addresses may be loaded and stored by the program (memory safety) and defines what the safety of code means. Our machine model and safety policy are trusted and are small enough to be "verifiable by inspection". A machine state (r, m) consists of a register bank r and a memory m, which are modeled as functions from numbers to contents (also numbers). A machine instruction is modeled by a relation between machine states (r, m) and (r', m') [11]. For example, a load instruction (`ld`) is specified by[2]

$$\texttt{ld}\ s, d \equiv \lambda r, m, r', m'.\ r'(d) = m(r(s)) \ \wedge\ \big(\forall x \neq d.\ r'(x) = r(x)\big) \ \wedge \\ m' = m \ \wedge\ \mathrm{readable}(r(s))$$

The machine operational semantics is modeled by a step relation $\mapsto$ that steps from one state (r, m) to another state (r', m') [11], where the state (r', m') is the result of first decoding the current machine instruction, incrementing the program counter and then executing the machine instruction.

The important property of our step relation is that it is deliberately partial: it omits any step that would be illegal under the safety policy. For example, suppose in some state (r, m) the program counter points to a `ld` instruction that would, if executed, load from an address that is unreadable according to the safety policy. Then, since our `ld` instruction requires that the address must be readable, there will not exist (r', m') such that $(r, m) \mapsto (r', m')$.

The mixing of machine semantics and safety policy is to follow the standard practice in type theory so that we can get a clean and uniform definition of code safety. For instance, we can define that a state is safe if it cannot lead to a stuck state.

$$\mathrm{safe}(r, m) \equiv \forall r', m'.\ (r, m) \mapsto^* (r', m') \ \Rightarrow \exists r'', m''.\ (r', m') \mapsto (r'', m'')$$

where $\mapsto^*$ denotes zero or more steps.

[2] Our step relation first increments pc, then executes an instruction. Thus, the semantics of `ld` does not include the semantics of incrementing the pc.

To show $\mathrm{safe}(r, m)$, it suffices to prove that the state is "safe for n steps," for any natural number n.

$$\mathrm{safe\_n}(n, r, m) \equiv \forall r', m'. \forall j < !n. (r, m) \mapsto^j (r', m') \Rightarrow \exists r'', m''. (r', m') \mapsto (r'', m'')$$

where $\mapsto^j$ denotes j steps being taken.

An assembly-language program C is a list of assembly instructions. For example,

$$C = \texttt{add r1, r1, r2; } \texttt{jmp } l_3\texttt{; } \texttt{ld [r2], r3; } \ldots$$

We use predicate $\mathrm{prog\_loaded}(m, C)$ to mean that code C is loaded in memory m:

$$\mathrm{prog\_loaded}(m, C) \equiv \forall\, 0 \leq k < |C|.\ \mathrm{decode}(m(4k), C_k)$$

where $|C|$ is the length of the list; predicate $\mathrm{decode}(x, C_k)$ means that word x is decoded into instruction C_k (the k-th instruction in C). In this paper, we assume code C is always loaded at start address 0 and thus the k-th instruction will be at address $4k$ in the memory (Sparc instructions are four bytes long).

We define that an assembly program C is safe if any initial state (r, m) satisfying the following is a safe state: code C is loaded inside m; the program counter initially points to address 0; when the program begins executing, entry condition ϕ_0[3] holds on the state (r, m).

$$\mathrm{safe\_code}(C) \equiv \forall r, m. \left(\mathrm{prog\_loaded}(m, C) \ \wedge\ r(\mathrm{pc}) = 0 \ \wedge\ (r, m) : \phi_0\right) \ \Rightarrow\ \mathrm{safe}(r, m)$$

3 Typed Assembly Language

In this section, we introduce a typed assembly language. We will show here a small subset of our actual implementation. Our full language has hundreds of operators and rules, as necessary for production-scale safety checking of real software (so it's a good thing that our full soundness proof is machine-checkable).

3.1 Syntax

Figure 1 shows our TAL syntax. A TAL program consists of assembly program C and type annotation Γ. Assembly program C is a sequence of assembly instructions, which include addition (`add`), load (`ld`) and branch always (`ba`).[4] Type annotation Γ takes the form of a label environment, which summarizes the preconditions of all the labels of the program. Our TAL has 32 registers, and labels are divisible by 4.

[3] In our implementation, the initial condition ϕ_0 is simple enough to be described directly in our underlying logic so that semantic model of types is not a part of the specification of the safety theorem; it is contained entirely within the *proof* of the theorem.

[4] Since our sample TAL cannot deal with delay slots, the `ba` instruction is really a `ba` followed by a `nop` in Sparc.

$$
\begin{array}{rl}
(\mathit{program}) & C ::= i \mid i; C \\
(\mathit{instruction}) & i ::= \mathtt{add}\, r, r, r \mid \mathtt{ld}\, r, r \mid \mathtt{ba}\, l \\
(\mathit{label\ env}) & \Gamma ::= \{l : \mathsf{codeptr}\,(\phi)\} \cap \ldots \\
(\mathit{register\ num}) & r ::= 0 \mid 1 \mid \ldots \mid 31 \\
(\mathit{label}) & l ::= 0 \mid 4 \mid 8 \mid \ldots
\end{array}
$$

Fig. 1. Syntax: Typed assembly language syntax

$$
\begin{array}{rl}
(\mathit{types}) & \tau ::= \mathsf{int} \mid \mathsf{int}_{=}(n) \\
& \mid \mathsf{box}\,(\tau) \mid \mathsf{codeptr}\,(\phi) \\
(\mathit{type\ env}) & \phi ::= \top_\phi \mid \bot_\phi \mid \{n : \tau\} \\
& \mid \phi_1 \cap \phi_2 \mid \phi[n \mapsto \tau] \\
(\mathit{nat}) & n ::= 0 \mid 1 \mid 2 \mid \ldots
\end{array}
$$

Fig. 2. Syntax: Types

Figure 2 lists the type and type environment constructors. They include integer type int and immutable reference type $\mathsf{box}\,(\tau)$. The language also has singleton type $\mathsf{int}_{=}(n)$ containing only value n. An address l has type $\mathsf{codeptr}\,(\phi)$ if it is safe to pass the control to address l provided that precondition ϕ is met.

A type environment ϕ specifies types of slots in a vector, such as a register bank or the list of program labels. Any vector satisfies environment $\top_\phi$, and no vector can satisfy $\bot_\phi$. A singleton environment $\{n : \tau\}$ means slot n (e.g., register n or label n) has type τ. Intersection type $\phi_1 \cap \phi_2$ can be used to type several slots of a vector, e.g. $\{n_1 : \tau_1\} \cap \{n_2 : \tau_2\}$ specifies that slots n_1 and n_2 have type τ_1 and τ_2 , respectively.

We use $\phi \subset \{n : \tau\}$ to describe that $\{n : \tau\}$ is one of the conjuncts in ϕ. We write $\phi(n)$ for the type of slot n in ϕ. Notation $\phi[n \mapsto \tau]$ updates the type of slot n to τ, by first removing the old entry for n in ϕ (if one exists), then intersecting it with $\{n : \tau\}$.

The type annotation Γ, or the label environment, specifies the type of each label in terms of the code pointer type, so it is also a type environment; we use the same operators for Γ as for ϕ.

3.2 Type Checking

There are three kinds of judgments in our type system:

- **Program** judgment $\vdash_{\mathtt{p}}\ C : \Gamma$ means that assembly program C is wellformed with respect to type annotation Γ.
- **Block** judgment $\Gamma; l\ \vdash_{\mathtt{b}}\ C : \Gamma'$ means that assembly program C, starting at address l, is wellformed with respect to Γ', assuming the global label-environment Γ. Environment Γ provides preconditions of labels to which C might jump; Environment Γ' is the collection of preconditions of labels inside C and is a part of Γ. Superficially, it seems semantically circular to judge the wellformedness of some labels (Γ') by assuming the wellformedness of all labels (Γ). However, as indicated in the overview section, the model of $\vdash_{\mathtt{b}}$ is that from a weaker assumption about Γ (every label inside is safe for k steps), we prove a stronger result about Γ' (every label inside is safe for $k+1$ steps).
- **Instruction** judgment $\Gamma; l\ \vdash_{\mathtt{i}}\ \{\phi_1\}\, i\, \{\phi_2\}$ means that assembly instruction i, at address l, is wellformed with respect to precondition ϕ_1 and postcondition ϕ_2. As in $\vdash_{\mathtt{b}}$, Γ provides label preconditions. The purpose of having location l in the judgment is to be able to compute the destination address for pc-relative jump instructions.

Typing rules except for instructions are shown in Figure 3. To check that program C is wellformed, the PROG rule will call $\Gamma; 0 \vdash_{\mathtt{b}} C : \Gamma$, thus recursively call BLOCK_I and BLOCK_2 rules to check that each basic block in C is wellformed.

Rule BLOCK_I first looks up precondition ϕ_1 and postcondition ϕ_2 in Γ for the current block, composed of one instruction i; checks the wellformedness of instruction i with respect to ϕ_1 and ϕ_2; then checks the rest of the code with respect to Γ'. Without loss of generality, we assume that each block has exactly one instruction. In rule BLOCK_2, the postcondition of the last instruction i is $\bot_\phi$, because the control is not allowed to be beyond the last instruction (an unconditional branch satisfies this postcondition).

$$\frac{\Gamma; 0 \vdash_{\mathtt{b}} C : \Gamma}{\vdash_{\mathtt{p}} C : \Gamma}\ \text{PROG} \qquad \frac{\begin{array}{c}\Gamma(l) = \mathsf{codeptr}\,(\phi_1) \qquad \Gamma(l+4) = \mathsf{codeptr}\,(\phi_2) \\ \Gamma; l \vdash_{\mathtt{i}} \{\phi_1\}\, i\, \{\phi_2\} \qquad \Gamma; l+4 \vdash_{\mathtt{b}} C : \Gamma'\end{array}}{\Gamma; l \vdash_{\mathtt{b}} i; C : \{l : \mathsf{codeptr}\,(\phi_1)\} \cap \Gamma'}\ \text{BLOCK\_1}$$

$$\frac{\Gamma(l) = \mathsf{codeptr}\,(\phi) \qquad \Gamma; l \vdash_{\mathtt{i}} \{\phi\}\, i\, \{\bot_\phi\}}{\Gamma; l \vdash_{\mathtt{b}} i : \{l : \mathsf{codeptr}\,(\phi)\}}\ \text{BLOCK\_2}$$

Fig. 3. Syntax: Typing rules (except for instructions)

Figure 4 shows typing rules for instructions. The rule for instruction `add` requires that the source registers are of type `int` beforehand and the destination register gets type `int` afterward. The rule for instruction `ba` needs to look up the type of the destination label through Γ and check the current precondition ϕ matches the destination one (A TAL usually has subtyping rules allowing the current precondition to be stronger than the destination one).

$$\frac{\phi \subset \{s_1 : \mathsf{int}\} \qquad \phi \subset \{s_2 : \mathsf{int}\}}{\Gamma; l \vdash_{\mathtt{i}} \{\phi\}\ \mathtt{add}\ s_1, s_2, d\, \{\phi[d \mapsto \mathsf{int}]\}} \qquad \frac{\phi \subset \{s : \mathsf{box}\,(\tau)\}}{\Gamma; l \vdash_{\mathtt{i}} \{\phi\}\ \mathtt{ld}\ s, d\, \{\phi[d \mapsto \tau]\}} \qquad \frac{\Gamma(l+d) = \mathsf{codeptr}\,(\phi)}{\Gamma; l \vdash_{\mathtt{i}} \{\phi\}\ \mathtt{ba}\ d\, \{\bot_\phi\}}$$

Fig. 4. Syntax: Typing rules for instructions

4 Indexed Model of Types

In this section, we give a brief description of the *indexed* model of types, which is introduced in [9] to model general recursive types. Our induction technique in the overview section is also inspired by the intuition behind the indexed model. In the indexed model, a type is a set of indexed values $\{\langle k, m, x\rangle\}$, where k is a natural number ("approximation" index), m is a memory, and x is an integer.

The indexed model of the types and type environments are listed below. For example, type $\mathsf{int}_{=}(3)$ would contain all the $\langle k, m, x\rangle$ such that x is 3. Memory m is a part of a value $\langle k, m, x\rangle$ because to express that x is of type $\mathsf{box}\,(\tau)$ we need to say that the content in the memory, or $m(x)$, is related to type τ.

$$\begin{aligned}
\mathsf{int} &\equiv \{\langle k, m, x\rangle \,|\, \text{true}\} & \mathsf{int}_{=}(n) &\equiv \{\langle k, m, x\rangle \,|\, x = n\}\\
\mathsf{box}\,(\tau) &\equiv \{\langle k, m, x\rangle \,|\, x \in \mathrm{dom}(m) \wedge \forall j < k.\, \mathrm{readable}(x) \wedge \langle j, m, m(x)\rangle \in \tau\}\\
\mathsf{codeptr}\,(\phi) &\equiv \{\langle k, m, x\rangle \,|\, \forall j, r.\, j < k \wedge r(\mathrm{pc}) = x \wedge (m, r) :_j \phi \Rightarrow \mathrm{safe_n}(j, r, m)\}
\end{aligned}$$

$$\begin{aligned}
\top_\phi &\equiv \{\langle k, m, \boldsymbol{x}\rangle \,|\, \text{true}\} & \bot_\phi &\equiv \{\langle k, m, \boldsymbol{x}\rangle \,|\, \text{false}\}\\
\{n : \tau\} &\equiv \{\langle k, m, \boldsymbol{x}\rangle \,|\, \langle k, m, x_n\rangle \in \tau\}\\
\phi_1 \cap \phi_2 &\equiv \{\langle k, m, \boldsymbol{x}\rangle \,|\, \langle k, m, \boldsymbol{x}\rangle \in \phi_1 \wedge \langle k, m, \boldsymbol{x}\rangle \in \phi_2\}\\
\phi[n \mapsto \tau] &\equiv \{\langle k, m, \boldsymbol{x}\rangle \,|\, \exists y.\, \langle k, m, \boldsymbol{x}[n \mapsto y]\rangle \in \phi \wedge \langle k, m, x_n\rangle \in \tau\}
\end{aligned}$$

We use $(m, x) :_k \tau$ as a syntactic sugar for $\langle k, m, x\rangle \in \tau$. We write $(m, x) : \tau$ to mean $(m, x) :_k \tau$ is true for any k, or (m, x) is a real member of type τ. Now we explain the purpose of index k in the model. In general, if $(m, x) :_k \tau$, value (m, x) may be a real member of type τ, or it may be a "fake" member that only k-approximately belongs to τ. Any program taking such a "fake" member as an input cannot tell the difference within k steps.

Let type τ be $\mathsf{box}\,(\mathsf{box}\,(\mathsf{int}))$. Suppose $(m, x) : \tau$, then x is a two-fold pointer and $m(m(x))$ is of type int. However, suppose we only know that $(m, x) : \mathsf{box}\,(\mathsf{int})$, then for one step (one dereference), (m, x) safely simulates membership in $\mathsf{box}\,(\mathsf{box}\,(\mathsf{int}))$. In this case, we can say $(m, x) :_1 \mathsf{box}\,(\mathsf{box}\,(\mathsf{int}))$.

One property of types is that they are closed under decreasing approximations, that is, if $(m, x) :_k \tau$ and $j < k$, then $(m, x) :_j \tau$.

Another example to understand the approximation index k is the type $\mathsf{codeptr}\,(\phi)$. A real member (m, l) of type $\mathsf{codeptr}\,(\phi)$ means that if condition ϕ is met, it is safe to jump to location l. Then $(m, l) :_k \mathsf{codeptr}\,(\phi)$ would mean that it is safe to execute k steps after jumping to l. Therefore, the definition of $\mathsf{codeptr}\,(\phi)$ says that for any j and r such that j is less than k, if the control is at location l and the current state satisfies ϕ, the state should be safe for j steps. In some sense, this definition only guarantees partial safety: safe within k steps. To show that location l is a safe location, we have to prove that it belongs to $\mathsf{codeptr}\,(\phi)$ under any k.

Sometimes we need to judge not only scalar values such as $\langle k, m, x\rangle$ but also vector values $\langle k, m, \boldsymbol{x}\rangle$ (a vector is a function from numbers to values). One use is to write $(m, r) :_k \phi$, which means that the contents of machine registers satisfy ϕ. In this case, $\boldsymbol{x}$ is the register bank r. Another use of vector types is the label environment Γ, which summarizes the types of all program labels. In this case, $\boldsymbol{x}$ would be the identity vector id (map l to l). For example, $(m, \mathrm{id}) : \{l : \mathsf{codeptr}\,(\phi)\}$ means that address l itself has type $\mathsf{codeptr}\,(\phi)$.

With the semantic model of types, all the subtyping rules and introduction/elimination rules of types (not shown in the paper) can be proved as lemmas [8,9]. In particular, the codeptr elimination rule CPTR_E is useful for the proof of Theorem 2 in Section 5.3.

$$\frac{(m, x) :_{k+1} \mathsf{codeptr}\,(\phi) \qquad r(pc) = x \qquad (m, r) :_k \phi}{\mathrm{safe\_n}(k, r, m)}\ \text{CPTR\_E}$$

5 Semantic Models of Typing Judgments

In this section, we develop models for typing judgments based on the induction technique in Section 1.2. From their models, each of the typing rules is proved as a derived lemma. Finally, the type-safety theorem is also proved as a derived lemma.

5.1 Subtype Induction

Our goal is to provide a proof that code C obeys our safety policy, or a proof of $\mathrm{safe\_code}(C)$, which means that any state containing the code C is safe arbitrarily many steps under condition ϕ_0 — exactly what the type $\mathsf{codeptr}\,(\phi_0)$ denotes. Thus, our goal is formalized as $(m, 0) : \mathsf{codeptr}\,(\phi_0)$, for any m containing code C.

As outlined in the overview section, we will prove a stronger result instead: all labels are of code-pointer types under corresponding preconditions. Formally, for any label l in the domain of label environment Γ, we will prove $(m, l) : \Gamma(l)$; another way to state this is $(m, \mathrm{id}) : \Gamma$.

A condition on m is that it must contain the code C. This condition can also be formalized as types. After all, in our model, types are predicates over states and can be used to specify invariants of states. Type $\mathsf{instr}\,(i)$ expresses that instruction i is in the memory.

$$\mathsf{instr}\,(i) \;\equiv\; \{\langle k, m, x\rangle \mid \mathrm{decode}(m(x), i)\}$$

Type environment constructor Δ turns a sequence of assembly instructions into a type environment that describes the code.

$$\Delta(i_0; i_1; \ldots) \;\equiv\; \{0 : \mathsf{instr}\,(i_0)\} \cap \{4 : \mathsf{instr}\,(i_1)\} \cap \ldots$$

With constructor Δ, judgment $(m, \mathrm{id}) : \Delta(C)$ formally states that memory m contains code C. For such a value (m, id), we want to show it also satisfies Γ, or $(m, \mathrm{id}) : \Gamma$. Now if we define

$$\Delta(C) \subset \Gamma \equiv \forall k, m.\ (m, \mathrm{id}) :_k \Delta(C) \;\Rightarrow (m, \mathrm{id}) :_k \Gamma$$

then $\Delta(C) \subset \Gamma$ expresses that any state containing code C respects invariants Γ under any approximation k.

We explore how to prove $\Delta(C) \subset \Gamma$. Assuming $(m, \mathrm{id}) :_k \Delta(C)$, we have to show $(m, \mathrm{id}) :_k \Gamma$. We will prove it by induction over k. When k is zero, judgment $(m, l) :_0 \Gamma(l)$ is trivially true since $\Gamma(l)$ is a code pointer type, which is always true at index zero by its definition. The inductive case is that, assuming $(m, \mathrm{id}) :_k \Delta(C) \Rightarrow (m, \mathrm{id}) :_k \Gamma$, we have to show that $(m, \mathrm{id}) :_{k+1} \Delta(C) \Rightarrow (m, \mathrm{id}) :_{k+1} \Gamma$. Since code environment $\Delta(C)$ ignores the index, $(m, \mathrm{id}) :_{k+1} \Delta(C)$ is equivalent to $(m, \mathrm{id}) :_k \Delta(C)$. Therefore, to prove $(m, \mathrm{id}) :_{k+1} \Gamma$, we have both $(m, \mathrm{id}) :_k \Delta(C)$ and $(m, \mathrm{id}) :_k \Gamma$.

Our intention is to give models to the typing judgments in our TAL based on this proof technique. To make the models concise, we abstract away from the indexes by defining a subtype-plus predicate $\Gamma_1 \Subset \Gamma_2$ to simulate the inductive case.

$$\Gamma_1 \Subset \Gamma_2 \equiv \forall k, m.\ (m, \mathrm{id}) :_k \Gamma_1 \;\Rightarrow (m, \mathrm{id}) :_{k+1} \Gamma_2$$

With the subtype-plus operator, the inductive case to prove $\Delta(C) \subset \Gamma$ can be written as $\Delta(C) \cap \Gamma \Subset \Gamma$: assuming code C is in the memory under index k and Γ is true under index k, prove that Γ is true under index $k+1$. The following subtype induction theorem formalizes what we have explained.

Theorem 1. *(Subtype Induction)*
$$\frac{\Delta(C) \cap \Gamma \Subset \Gamma}{\Delta(C) \subset \Gamma}$$

5.2 Semantic Model of Typing Judgments

At the heart of our semantic model is a set of concise definitions for the typing judgments based on the abstractions (especially $\Subset$) we have developed. We hereby exhibit such definitions:

$$\begin{array}{ll} \vdash_{\mathtt{p}} C : \Gamma & \equiv \Delta(C) \subset \Gamma \\ \Gamma; l \vdash_{\mathtt{b}} C : \Gamma' & \equiv \mathsf{offset}_l(\Delta(C)) \cap \Gamma \Subset \Gamma' \\ \Gamma; l \vdash_{\mathtt{i}} \{\phi_1\}\, i\, \{\phi_2\} & \equiv \{l : \mathsf{instr}\,(i)\} \cap \Gamma \cap \{l+4 : \mathsf{codeptr}\,(\phi_2)\} \Subset \{l : \mathsf{codeptr}\,(\phi_1)\} \end{array}$$

where $\mathsf{offset}_l(\{0 : \tau_1\} \cap \{4 : \tau_4\} \cap \ldots) = \{l : \tau_1\} \cap \{l+4 : \tau_4\} \cap \ldots$.

The model of $\vdash_{\mathtt{p}} C : \Gamma$ means that any state having the code inside respects invariants Γ.

The judgment $\Gamma; l \vdash_{\mathtt{b}} C : \Gamma'$ judges the validity of Γ' by assuming Γ. Since Γ' is the collection of preconditions of labels inside C and is a part of Γ, the judgment itself has a superficial semantic circularity. We solve this circularity by giving it a model based on operator $\Subset$. By assuming Γ to approximation k, we prove Γ' to approximation $k+1$. Also, since code C starts at address l, we need to use $\mathsf{offset}_l(\Delta(C))$ to make a code environment that starts at address l.

The semantics of $\Gamma; l \vdash_{\mathtt{i}} \{\phi_1\}\, i\, \{\phi_2\}$ follows the same principle as the one for the model of $\vdash_{\mathtt{b}}$. We assume that instruction i is at location l and Γ holds to approximation k, we prove location l is a code pointer to approximation $k+1$. In the model of $\vdash_{\mathtt{i}}$, we have an extra assumption $\{l+4 : \mathsf{codeptr}\,(\phi_2)\}$. In our sample TAL, since every basic block has only one instruction, every address would have a precondition in Γ. Therefore, $\{l+4 : \mathsf{codeptr}\,(\phi_2)\}$ is a part of Γ and need not have been specified as a separate conjunct. However, in the case of multi-instruction basic blocks, Γ would only have preconditions for each basic block, ϕ_2 in this case would be reconstructed by the type system and not available in Γ.

5.3 Semantic Proofs of Typing Rules

Using these models, we can prove both the type-safety theorem and the typing rules in Fig.3.

Theorem 2. *(Type Safety)*
$$\frac{\vdash_{\mathtt{p}} C : \Gamma \qquad \Gamma \subset \{0 : \mathsf{codeptr}\,(\phi_0)\}}{\mathrm{safe\_code}(C)}$$

Proof. From the definition of $\mathrm{safe\_code}(C)$, we have the following assumptions for state (r, m) and want to show that $\mathrm{safe}(r, m)$.

$$i)\ \mathrm{prog\_loaded}(C, m) \qquad ii)\ r(pc) = 0 \qquad iii)\ (m, r) : \phi_0$$

On the other hand, the model of $\vdash_{\mathtt{p}}\ C : \Gamma$ is $\Delta(C) \subset \Gamma$. The deduction steps from $\Delta(C) \subset \Gamma$ to $\mathrm{safe}(r, m)$ are summarized by the following proof tree.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathrm{prog\_loaded}(C,m)}{(m,\mathrm{id}) : \Delta(C)}\ (7) \quad \Delta(C) \subset \Gamma \qquad \Gamma \subset \{0 : \mathsf{codeptr}\,(\phi_0)\}}{(m,\mathrm{id}) : \{0 : \mathsf{codeptr}\,(\phi_0)\}}\ (6)}{(m,0) : \mathsf{codeptr}\,(\phi_0)}\ (5)}{\forall k.\ (m,0) :_k \mathsf{codeptr}\,(\phi_0)}\ (4)}{\forall k.\ (m,0) :_{k+1} \mathsf{codeptr}\,(\phi_0)}\ (3a) \qquad \dfrac{(m,r) : \phi_0}{\forall k.\ (m,r) :_k \phi_0}\ (3b) \qquad r(pc) = 0}{\dfrac{\forall k.\ \mathrm{safe\_n}(k, r, m)}{\mathrm{safe}(r, m)}\ (1)}\ (2)$$

Step (1) says to prove (r, m) is safe, it suffices to prove that (r, m) is safe for an arbitrary k steps. Step (2) is justified by rule CPTR_E in Section 4. Step (3a) is by universal instantiation. Step (4) is just the unfolding of the syntactic sugar of $(m, 0) : \mathsf{codeptr}\,(\phi_0)$. Step (5) is by the definition of the singleton environment. Step (6) is by the transitivity of subtyping. Step (7) can be easily proved by unfolding definitions. □

Theorem 3.
$$\dfrac{\Gamma; 0\ \vdash_{\mathtt{b}}\ C : \Gamma}{\vdash_{\mathtt{p}}\ C : \Gamma}\ \textsc{prog}$$

Proof. From the models of $\vdash_{\mathtt{p}}$ and $\vdash_{\mathtt{b}}$, we have to prove $\Delta(C) \subset \Gamma$ from $\Delta(C) \cap \Gamma \Subset \Gamma$ — exactly the subtype induction theorem (Theorem 1). □

Theorem 4.
$$\dfrac{\Gamma(l) = \mathsf{codeptr}\,(\phi_1) \qquad \Gamma(l+4) = \mathsf{codeptr}\,(\phi_2) \quad \Gamma; l\ \vdash_{\mathtt{i}}\ \{\phi_1\}\, i\, \{\phi_2\} \qquad \Gamma; l+4\ \vdash_{\mathtt{b}}\ C : \Gamma'}{\Gamma; l\ \vdash_{\mathtt{b}}\ i; C : \{l : \mathsf{codeptr}\,(\phi_1)\} \cap \Gamma'}\ \textsc{block\_1}$$

Proof. The models of $\Gamma; l\ \vdash_{\mathtt{i}}\ \{\phi_1\}\, i\, \{\phi_2\}$[5] and $\Gamma; l+4\ \vdash_{\mathtt{b}}\ C : \Gamma'$ gives us that

$$\{l : \mathsf{instr}\,(i)\} \cap \Gamma \Subset \{l : \mathsf{codeptr}\,(\phi_1)\} \qquad \mathsf{offset}_{l+4}(\Delta(C)) \cap \Gamma \Subset \Gamma'$$

The goal $\mathsf{offset}_l(\Delta(i; C)) \cap \Gamma \Subset \{l : \mathsf{codeptr}\,(\phi_1)\} \cap \Gamma'$ is proved by the following lemmas:

$$\dfrac{\Gamma \Subset \Gamma_1 \qquad \Gamma \Subset \Gamma_2}{\Gamma \Subset \Gamma_1 \cap \Gamma_2} \qquad \mathsf{offset}_l(\Delta(i; C)) = \{l : \mathsf{instr}\,(i)\} \cap \mathsf{offset}_{l+4}(\Delta(C))$$

□

The proof of rule BLOCK_2 is similar.

[5] The model of $\vdash_{\mathtt{i}}$ has another clause $\{l + 4 : \mathsf{codeptr}\,(\phi_2)\}$ on the left of $\Subset$. However, since $\Gamma(l + 4) = \mathsf{codeptr}\,(\phi_2)$, we can prove that $\Gamma \cap \{l + 4 : \mathsf{codeptr}\,(\phi_2)\} = \Gamma$.

5.4 Semantic Proofs of Machine Instructions

What remains is the proofs of typing rules for instructions (Fig.4). We will show the technique by informally proving the LOAD rule.

$$\frac{\phi \subset \{s : \mathsf{box}\,(\tau)\}}{\Gamma; l \vdash_{\mathtt{i}} \{\phi\}\,\mathtt{ld}\ s, d\,\{\phi[d \mapsto \tau]\}}\ \text{LOAD}$$

The precondition states that register s is a pointer to type τ; the postcondition is that register d gets type τ and types of other registers remain the same. This typing rule is intuitively "correct" since operationally $\mathtt{ld}\ s, d$ loads the content at address $r(s)$ in the memory into register d. The semantic model of $\vdash_{\mathtt{i}}$ tells us what the "correctness" of the rule LOAD means:

$$\{l : \mathsf{instr}\,(\mathtt{ld}\ s, d)\} \cap \Gamma \cap \{l+4 : \mathsf{codeptr}\,(\phi')\} \Subset \{l : \mathsf{codeptr}\,(\phi)\}$$

where $\phi' = \phi[d \mapsto \tau]$. That is, forall k and m, we should prove $(m, l) :_{k+1} \mathsf{codeptr}\,(\phi)$, or location l is safe within $k+1$ steps. We can assume (1) there is an instruction $\mathtt{ld}\ s, d$ at address l in m; (2) all the labels in Γ are code pointers to approximation k; (3) label $l+4$ is of type $\mathsf{codeptr}\,(\phi')$ to approximation k.

By the definition of $(m, l) :_{k+1} \mathsf{codeptr}\,(\phi)$, we start at (r, m) with condition ϕ met and control at l. By the semantics of $\mathtt{ld}\ s, d$, if the location (location $r(s)$) to read is readable, we can find a succeeding state (r', m') such that $(r, m) \mapsto (r', m')$. Not by coincidence, rule LOAD has a premise that $\phi \subset \{s : \mathsf{box}\,(\tau)\}$; together with that ϕ is met on state (r, m), $\mathrm{readable}(r(s))$ can be shown.

Therefore, there is a state (r', m') that (r, m) can step to because of the execution of the instruction $\mathtt{ld}\ s, d$. By the semantics of $\mathtt{ld}$, condition ϕ' can be proved to hold on state (r', m') and the control of (r', m') is at label $l+4$. Because label $l+4$ is of type $\mathsf{codeptr}\,(\phi')$ to approximation k, state (r', m') is safe within k steps. Taking the step by $\mathtt{ld}$ into account, the first state (r, m) is safe within $k+1$ steps.

Proofs for other typing rules of instructions follow the same scheme. In the case of control-transfer instructions like $\mathtt{ba}$, the assumption (2) about Γ guarantees that it is safe to jump to the destination address.

6 Implementation

Our work is a part of the Foundational Proof-Carrying Code project [12], which includes a compiler from ML to Sparc, a typed assembly language called LTAL [6], and semantic proofs of typing rules. We have successfully given models to typing judgments in LTAL based on proof techniques in this paper, with a proved type-safety theorem and nearly complete semantic proofs of the typing rules. All the proofs are written and machine-checked in a theorem-proving system — Twelf [13]: 1949 lines of axioms of the logic, arithmetic, and the specification of the SPARC machine; 23562 lines of lemmas of logic and arithmetic, theories of mathematical foundations such as sets and lists; 72036 lines of lemmas about conventions of machine states and semantic model of types; 18895 lines of lemmas about machines instructions and the LTAL calculus. Most of the incomplete semantic proofs are about machines instruction semantics.

To focus the presentation on the essential ideas, we have not shown many features of our actual implementation. In this paper we have used immutable reference types to describe data structures in the memory and proved that programs can safely access these data structures. Our implementation can also deal with mutable references [10] so that programs can safely update data structures in the memory. Allocation of new data structures in the memory have also been taken into account by following the allocation model of SML/NJ. However, we do not currently support either explicit deallocation (`free`) or garbage collection. LTAL is also more expressive, including type variables, quantified types and condition-code types. It can also type-check position-independent code and multi-instruction basic blocks. Our semantic model supports all these features.

7 Related Work

There has been some work in the program-verification community to use a semantic approach to prove Hoare-logic rules as lemmas in an underlying logic [7,14,15]. Such proofs have been mechanized in HOL [14]. These works use first-order or higher-order logic to specify the invariants and have the difficulty that loop invariants cannot be derived automatically, so the approach does not scale to large programs.

Hamid *et al.* [16] and Crary [17] use a syntactic approach to prove type soundness. The first stage of their approach develops a typed assembly language, which is also given an operational semantics on an abstract machine. Then syntactic type-soundness theorems are proved on this abstract machine following the scheme presented by Wright and Felleisen [18]. The second stage uses a simulation relation between the abstract machine and the concrete architecture. The syntactic approach does not need the building of denotational semantics for complicated types such as recursive types, and it can also have machine-checkable proofs. However, the simulation step between the abstract machine and a full-fledged architecture is not a trivial task.

In some sense, the problem we solve in this paper is to give models to unstructured programs with goto statements and labels. There has been work by de Bruin [19] to give goto statements a domain-theoretic model. His approach to prove that code respects invariants is by approximations over code behavior, i.e. any k-th approximations of code behavior respects invariants. In our approach, we use types as invariants and do approximations over types, i.e. code respects any k-th approximations of types.

In conclusion, we have shown how to build end-to-end foundational safety proofs of programs on a real machine. We have constructed a semantic model for typing judgments of a typed assembly language and given proofs for both the type-safety theorem and typing rules. Our approach allows a typing derivation to be interpreted as a machine-checkable safety proof at the machine level.

References

1. Necula, G.: Proof-carrying code. In: 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New York, ACM Press (1997) 106–119
2. Morrisett, G., Walker, D., Crary, K., Glew, N.: From System F to typed assembly language. ACM Trans. on Programming Languages and Systems **21** (1999) 527–568

3. Morrisett, G., Crary, K., Glew, N., Grossman, D., Samuels, R., Smith, F., Walker, D., Weirich, S., Zdancewic, S.: TALx86: A realistic typed assembly language. In: Second ACM SIGPLAN Workshop on Compiler Support for System Software, Atlanta, GA (1999) 25–35 INRIA Technical Report 0288, March 1999.
4. League, C., Shao, Z., Trifonov, V.: Precision in practice: A type-preserving Java compiler. In: Proc. Int'l. Conf. on Compiler Construction. (2003)
5. Colby, C., Lee, P., Necula, G.C., Blau, F., Cline, K., Plesko, M.: A certifying compiler for Java. In: Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00), New York, ACM Press (2000)
6. Chen, J., Wu, D., Appel, A.W., Fang, H.: A provably sound TAL for back-end optimization. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '03). (2003) 208–219
7. Schmidt, D.A.: Denotational Semantics: A Methodology for Language Development. Allyn and Bacon, Boston (1986)
8. Appel, A.W., Felty, A.P.: A semantic model of types and machine instructions for proof-carrying code. In: POPL '00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press (2000) 243–253
9. Appel, A.W., McAllester, D.: An indexed model of recursive types for foundational proof-carrying code. ACM Trans. on Programming Languages and Systems **23** (2001) 657–683
10. Ahmed, A., Appel, A.W., Virga, R.: A stratified semantics of general references embeddable in higher-order logic. In: 17th Annual IEEE Symposium on Logic in Computer Science (LICS 2002). (2002) 75–86
11. Michael, N.G., Appel, A.W.: Machine instruction syntax and semantics in higher-order logic. In: 17th International Conference on Automated Deduction, Berlin, Springer-Verlag (2000) 7–24 LNAI 1831.
12. Appel, A.W.: Foundational proof-carrying code. In: Symposium on Logic in Computer Science (LICS '01), IEEE (2001) 247–258
13. Pfenning, F., Schürmann, C.: System description: Twelf — a meta-logical framework for deductive systems. In: The 16th International Conference on Automated Deduction, Berlin, Springer-Verlag (1999)
14. Gordon, M.: Mechanizing programming logics in higher-order logic. In G.M. Birtwistle, P.A. Subrahmanyam, eds.: Current Trends in Hardware Verification and Automatic Theorem Proving, Banff, Canada, Springer-Verlag, Berlin (1988) 387–439
15. Wahab, M.: Verification and abstraction of flow-graph programs with pointers and computed jumps. Research Report CS-RR-354, Department of Computer Science, University of Warwick, Coventry, UK (1998)
16. Hamid, N., Shao, Z., Trifonov, V., Monnier, S., Ni, Z.: A syntactic approach to foundational proof-carrying code. In: Proc. 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02). (2002) 89–100
17. Crary, K.: Toward a foundational typed assembly language. In: The 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press (2003) 198–212
18. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and Computation **115** (1994) 38–94
19. de Bruin, A.: Goto statements: Semantics and deduction systems. Acta Informatica **15** (1981) 385–424

Rule-Based Runtime Verification

Howard Barringer*[1], Allen Goldberg[2], Klaus Havelund[2], and Koushik Sen**[3]

[1] University of Manchester, England
[2] Kestrel Technology, NASA Ames Research Center, USA
[3] University of Illinois, Urbana Champaign, USA

Abstract. We present a rule-based framework for defining and implementing finite trace monitoring logics, including future and past time temporal logic, extended regular expressions, real-time logics, interval logics, forms of quantified temporal logics, and so on. Our logic, EAGLE, is implemented as a Java library and involves novel techniques for rule definition, manipulation and execution. Monitoring is done on a state-by-state basis, without storing the execution trace.

1 Introduction

Runtime verification, or runtime monitoring, comprises having a software module, an observer, monitor the execution of a program and check its conformity with a requirement specification, often written in a temporal logic or as a state machine. Runtime verification can be applied to automatically evaluate test runs, either on-line, or off-line analyzing stored execution traces; or it can be used on-line during operation, potentially steering the application back to a safety region if a property is violated. It is highly scalable. Several runtime verification systems have been developed, of which some were presented at three recent international workshops on runtime verification [1].

Linear temporal logic (LTL) [19] has been core to several of these attempts. The commercial tool Temporal Rover (TR) [6,7] supports a fixed future and past time LTL, with the possibility of specifying real-time and data constraints (time-series) as annotations on the temporal operators. Its implementation is based on alternating automata. Algorithms using alternating automata to monitor LTL properties are also proposed in [9], and a specialized LTL collecting statistics along the execution trace is described in [8]. The MAC logic [18] is a form of past time LTL with operators inspired by interval logics and which models real-time via explicit clock variables. A logic based on extended regular expressions [20] has also been proposed and is argued to be more succinct for certain properties. The logic described in [16] is a sophisticated interval logic, argued to be more user-friendly than plain LTL. Our own previous work includes the development of several algorithms, such as generating dynamic programming algorithms for past time logic [14], using a rewriting system for monitoring future time logic [12,13], or generating Büchi automata inspired algorithms adapted to finite trace LTL [11].

* This author is most grateful to RIACS/USRA and to the UK's EPSRC under grant GR/S40435/01 for the partial support provided to conduct this research whilst at NASA Ames Research Center.

** This author is grateful for the support received from RIACS to undertake this research while participating in the Summer Student Research Program at the NASA Ames Research Center.

B. Steffen and G. Levi (Eds.): VMCAI 2004, LNCS 2937, pp. 44–57, 2004.

This large variety of logics prompted us to search for a small and general framework for defining monitoring logics, which would be powerful enough to capture essentially all of the above described logics, hence supporting future and past time logics, interval logics, extended regular expressions, state machines, real-time and data constraints, and statistics. The framework should support the definition of new logics in an easy manner and should support the monitoring of programs with their complex program states. The result of our search is the logic EAGLE presented in this paper. The EAGLE logic and its implementation for runtime monitoring has been significantly influenced by earlier work on the executable, trace generating as well as trace checking, temporal logic METATEM [3]. In METATEM a linear-time temporal formula is separated [10] into a boolean combination of pure past, present and pure future time formulas. Conditioned by the past, the present-time, or state, formulas determine how the state for the current moment in time is built and the pure future time formulas yield obligations that need to be fulfilled at some time later. The separation result, rules and future obligations are central in our current work. However, the fundamental difference between METATEM and EAGLE is that the METATEM interpreter builds traces state by state, whereas EAGLE is used for checking given finite traces: costly implementation features, such as backtracking and loop-checking, are not required.

We recently discovered parallel work [17] using recursive equations to implement a real-time logic. However we had already developed the ideas further. We provide the language of recursive equations to the user, we support a mixture of future time and past time operators, we treat real-time as a special case of data values, and hence we allow a very general logic for reasoning about data, including the possibility of relating data values across the execution trace, both forwards and backwards.

This paper is structured as follows. Section 2 introduces our logic framework. In section 3 we discuss the algorithm and calculus that underlies our implementation, which is then briefly described along with initial experimentation in section 4. Further papers on EAGLE are available covering material that couldn't be covered in this paper. In [4], we illustrate that when EAGLE is specialized to a propositional LTL our monitoring algorithm is space efficient with an upper bound of $O(m^2 \log m 2^m)$, where m is the size of the monitored formula; in [5], we present full details of our current Java-based monitoring algorithm and its associated rewrite calculus.

2 The Logic

In this section we introduce our temporal finite trace monitoring logic EAGLE. The logic offers a succinct but powerful set of primitives, essentially supporting recursive parameterized equations, with a minimal/maximal fix-point semantics together with three temporal operators: next-time, previous-time, and concatenation. The next-time and previous-time operators can be used for defining future time respectively past time temporal logics on top of EAGLE. The concatenation operator can be used to define interval logics and an extended regular expression language. Rules can be parameterized with formulas, and with data to allow for the expression of data constraints, including real-time constraints. Atomic propositions are boolean expressions over a program state, Java states in the current implementation. The logic is first introduced informally through two examples whereafter its syntax and semantics is given. Finally, its relationship to some other important logics is outlined.

2.1 EAGLE by Example

Fundamental Concepts. Assume we want to state a property about a program P, which contains the declaration of two integer variables x and y. We want to state that whenever x is positive then eventually y becomes positive. The property can be written as follows in classical future time LTL: $\Box(x > 0 \rightarrow \Diamond y > 0)$. The formulas $\Box F$ (always F) and $\Diamond F$ (eventually F), for some property F, usually satisfy the following equivalences, where the temporal operator $\bigcirc F$ stands for *next F* (meaning '*in next state F*'):

$$\Box F \equiv F \wedge \bigcirc(\Box F) \qquad \Diamond F \equiv F \vee \bigcirc(\Diamond F)$$

One can for example show that $\Box F$ is a solution to the recursive equivalence $X \equiv F \wedge \bigcirc X$; in fact it is the maximal solution[1]. A fundamental idea in our logic is to support this kind of recursive definition, and to enable users define their own temporal combinators using equations similar to those above. In the current framework one can write the following definitions for the two combinators `Always` and `Eventually`, and the formula to be monitored (M_1):

$$\begin{aligned}
&\underline{\max}\ \texttt{Always}(\underline{\text{Form}}\ F) = F \wedge \bigcirc \texttt{Always}(F)\\
&\underline{\min}\ \texttt{Eventually}(\underline{\text{Form}}\ F) = F \vee \bigcirc \texttt{Eventually}(F)\\
&\underline{\text{mon}}\ M_1 = \texttt{Always}(x > 0 \rightarrow \texttt{Eventually}(y > 0))
\end{aligned}$$

The `Always` operator is defined as having a maximal fix-point interpretation; the `Eventually` operator is defined as having a minimal interpretation. Maximal rules define safety properties (nothing bad ever happens), while minimal rules define liveness properties (something good eventually happens). For us, the difference only becomes important when evaluating formulas at the boundaries of a trace. To understand how this works it suffices to say here that monitored rules evolve as new states are appearing. Assume that the end of the trace has been reached (we are beyond the last state) and a monitored formula F has evolved to F'. Then all applications in F' of maximal fix-point rules will evaluate to true, since they represent safety properties that apparently have been satisfied throughout the trace, while applications of minimal fix-point rules will evaluate to false, indicating that some event did not happen. Assume for example that we evaluate the formula M_1 in a state where $x > 0$ and $y \leq 0$, then as a liveness obligation for the future we will have the expression:

$$\texttt{Eventually}(y > 0) \wedge \texttt{Always}(x > 0 \rightarrow \texttt{Eventually}(y > 0))$$

Assume that we at this point detect the end of the trace; that is: we are beyond the last state. The outstanding liveness obligation $\texttt{Eventually}(y > 0)$ has not yet been fulfilled, which is an error. This is captured by the evaluation of the minimal fix-point combinator `Eventually` being false at this point. The remaining other obligation from the $\wedge$-formula, namely, $\texttt{Always}(x > 0 \rightarrow \texttt{Eventually}(y > 0))$, is a safety property and evaluates to true.

For completeness we provide remaining definitions of the future time LTL operators $\mathcal{U}$ (until) and $\mathcal{W}$ (unless) below. Note how $\mathcal{W}$ is defined in terms of other operators. However, it could have been defined recursively.

$$\begin{aligned}
&\underline{\min}\ \texttt{Until}(\underline{\text{Form}}\ F_1, \underline{\text{Form}}\ F_2) = F_2 \vee (F_1 \wedge \bigcirc \texttt{Until}(F_1, F_2))\\
&\underline{\max}\ \texttt{Unless}(\underline{\text{Form}}\ F_1, \underline{\text{Form}}\ F_2) = \texttt{Until}(F_1, F_2) \vee \texttt{Always}(F_1)
\end{aligned}$$

[1] Similarly, $\Diamond F$ is a *minimal* solution to the equivalence $X \equiv F \vee \bigcirc X$

Data Parameters. We have seen how rules can be parameterized with formulas. Let us modify the above example to include data parameters. Suppose we want to state the property: *"whenever at some point $x = k > 0$ for some k, then eventually $y = k$"*. This can be expressed as follows in quantified LTL: $\Box(x > 0 \rightarrow \exists k.(x = k \wedge \Diamond y = k))$. We use a parameterized rule to state this property, capturing the value of x when $x > 0$ as a rule parameter.

$$\underline{\text{min}}\ \texttt{R}(\underline{\text{int}}\ k) = \texttt{Eventually}(y = k) \qquad \underline{\text{mon}}\ M_2 = \texttt{Always}(x > 0 \rightarrow \texttt{R}(x))$$

Rule R is parameterized with an integer k, and is instantiated in M_2 when $x > 0$, hence capturing the value of x at that moment. Rule R replaces the existential quantifier. The logic also provides a previous-time operator, which allows us to define past time operators; the data parametrization works uniformly for rules over past as well as future, which is non-trivial to achieve since the implementation does not store the trace, see Section 4. Data parametrization is also used to elegantly model real-time logics.

2.2 Syntax and Semantics

Syntax. A specification S consists of a declaration part D and an observer part O. D consists of zero or more rule definitions R, and O consists of zero or more monitor definitions M, which specify what to be monitored. Rules and monitors are named (N).

$$\begin{array}{lll}
S & ::= & D\ O \\
D & ::= & R^* \\
O & ::= & M^* \\
R & ::= & \{\underline{\text{max}} \mid \underline{\text{min}}\}\ N(T_1\ x_1, \ldots, T_n\ x_n) = F \\
M & ::= & \underline{\text{mon}}\ N = F \\
T & ::= & \underline{\text{Form}} \mid \textit{primitive type} \\
F & ::= & \textit{expression} \mid \underline{\text{true}} \mid \underline{\text{false}} \mid \neg F \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid F_1 \rightarrow F_2 \mid \\
 & & \bigcirc F \mid \odot F \mid F_1 \cdot F_2 \mid N(F_1, \ldots, F_n) \mid x_i
\end{array}$$

A rule definition R is preceded by a keyword indicating whether the interpretation is maximal or minimal (which we recall determines the value of a rule application at the boundaries of the trace). Parameters are typed, and can either be a formula of type Form, or of a primitive type, such as int, long, float, etc.. The body of a rule/monitor is a boolean valued formula of the syntactic category $Form$ (with meta-variables F, etc.). Any recursive call on a rule must be strictly guarded by a temporal operator. The propositions of this logic are boolean expressions over an observer state. Formulas are composed using standard propositional logic operators together with a next-state operator ($\bigcirc F$), a previous-state operator ($\odot F$), and a concatenation-operator ($F_1 \cdot F_2$). Finally, rules can be applied and their arguments must be type correct. That is, an argument of type Form can be any formula, with the restriction that if the argument is an expression, it must be of boolean type. An argument of a primitive type must be an expression of that type. Arguments can be referred to within the rule body (x_i).

In what follows, a rule N of the form

$$\{\underline{\text{max}}|\underline{\text{min}}\}\ N(\underline{\text{Form}}\ f_1, \ldots, \underline{\text{Form}}\ f_m, T_1\ p_1, \ldots, T_n\ p_n) = B,$$

where $f_1, \ldots f_m$ are arguments of type $\underline{\text{Form}}$ and $p_1, \ldots p_n$ are arguments of primitive type, is written in short as

$$\{\underline{\text{max}}|\underline{\text{min}}\}\ N(\overline{\underline{\text{Form}}}\ \overline{f}, \overline{T}\ \overline{p}) = B$$

where $\overline{f}$ and $\overline{p}$ represent tuples of type $\overline{\underline{\text{Form}}}$ and $\overline{T}$ respectively. Without loss of generality, in the above rule we assume that all the arguments of type $\underline{\text{Form}}$ appear first.

Semantics. The semantics of the logic is defined in terms of a satisfaction relation $\models$ between execution traces and specifications. An execution trace σ is a finite sequence of program states $\sigma = s_1 s_2 \ldots s_n$, where $|\sigma| = n$ is the length of the trace. The i'th state s_i of a trace σ is denoted by $\sigma(i)$. The term $\sigma^{[i,j]}$ denotes the sub-trace of σ from position i to position j, both positions included; if $i \geq j$ then $\sigma^{[i,j]}$ denotes the empty trace. In the implementation a state is a user defined Java object that is updated through a user provided *updateOnEvent* method for each new event generated by the program. Given a trace σ and a specification $D\ O$, satisfaction is defined as follows:

$$\sigma \models D\ O \ \text{ iff } \ \forall\, (\underline{\text{mon}}\ N = F) \in O\ .\ \sigma, 1 \models_D F$$

That is, a trace satisfies a specification if the trace, observed from position 1 (the first state), satisfies each monitored formula. The definition of the satisfaction relation $\models_D \subseteq (\mathit{Trace} \times \mathbf{nat}) \times \underline{\text{Form}}$, for a set of rule definitions D, is presented below, where $0 \leq i \leq n+1$ for some trace $\sigma = s_1 s_2 \ldots s_n$. Note that the position of a trace can become 0 (before the first state) when going backwards, and can become $n+1$ (after the last state) when going forwards, both cases causing rule applications to evaluate to either true if maximal or false if minimal, without considering the body of the rules at that point.

$$
\begin{array}{lll}
\sigma, i \models_D \mathit{expression} & \text{iff} & 1 \leq i \leq |\sigma| \text{ and } \mathit{evaluate}(\mathit{expression})(\sigma(i)) == \mathit{true} \\
\sigma, i \models_D \underline{\text{true}} & & \\
\sigma, i \not\models_D \underline{\text{false}} & & \\
\sigma, i \models_D \neg F & \text{iff} & \sigma, i \not\models_D F \\
\sigma, i \models_D F_1 \wedge F_2 & \text{iff} & \sigma, i \models_D F_1 \text{ and } \sigma, i \models_D F_2 \\
\sigma, i \models_D F_1 \vee F_2 & \text{iff} & \sigma, i \models_D F_1 \text{ or } \sigma, i \models_D F_2 \\
\sigma, i \models_D F_1 \rightarrow F_2 & \text{iff} & \sigma, i \models_D F_1 \text{ implies } \sigma, i \models_D F_2 \\
\sigma, i \models_D \bigcirc F & \text{iff} & i \leq |\sigma| \text{ and } \sigma, i+1 \models_D F \\
\sigma, i \models_D \bigodot F & \text{iff} & 1 \leq i \text{ and } \sigma, i-1 \models_D F \\
\sigma, i \models_D F_1 \cdot F_2 & \text{iff} & \exists j \text{ s.t. } i \leq j \leq |\sigma|+1 \text{ and } \sigma^{[1,j-1]}, i \models_D F_1 \text{ and } \sigma^{[j,|\sigma|]}, 1 \models_D F_2 \\
\sigma, i \models_D N(\overline{F}, \overline{P}) & \text{iff} & \left\{ \begin{array}{l} \text{if } 1 \leq i \leq |\sigma| \text{ then:} \\ \quad \sigma, i \models_D B[\overline{f} \mapsto \overline{F}, \overline{p} \mapsto \mathit{evaluate}(\overline{P})(\sigma(i))] \\ \quad \text{where } (N(\overline{\underline{\text{Form}}}\ \overline{f}, \overline{T}\ \overline{p}) = B)\ \in D \\ \text{otherwise, if } i = 0 \text{ or } i = |\sigma|+1 \text{ then:} \\ \quad \text{rule } N \text{ is defined as } \underline{\text{max}} \text{ in } D \end{array} \right.
\end{array}
$$

An expression (a proposition) is evaluated in the current state in case the position i is within the trace ($1 \leq i \leq n$). In the boundary cases ($i = 0$ and $i = n+1$) a proposition evaluates to false. Propositional operators have their standard semantics in all positions. A next-time formula $\bigcirc F$ evaluates to true if the current position is not beyond the

last state and F holds in the next position. Dually for the previous-time formula. The concatenation formula $F_1 \cdot F_2$ is true if the trace σ can be split into two sub-traces $\sigma = \sigma_1\sigma_2$, such that F_1 is true on σ_1, observed from the current position i, and F_2 is true on σ_2 (ignoring σ_1, and thereby limiting the scope of past time operators). Applying a rule within the trace (positions $1 \ldots n$) consists of replacing the call with the right-hand side of the definition, substituting arguments for formal parameters; if an argument is of primitive type its evaluation in the current state is substituted for the associated formal parameter of the rule, thereby capturing a desired freeze variable semantics. At the boundaries (0 and $n+1$) a rule application evaluates to true if and only if it is maximal.

2.3 Relationship to Other Logics

The logical system defined above is expressively rich; indeed, any linear-time temporal logic, whose temporal modalities can be recursively defined over the next, past or concatenation modalities, can be embedded within it. Furthermore, since in effect we have a limited form of quantification over possibly infinite data sets, and concatenation, we are strictly more expressive than, say, a linear temporal fixed point logic (over next and previous). A formal characterization of the logic is beyond the scope of this paper, however, we demonstrate the logic's utility and expressiveness through examples.

Past Time LTL: A past time linear temporal logic, i.e. one whose temporal modalities only look to the past, could be defined in the mirror way to the future time logic exemplified in the introduction by using the built-in previous modality, $\bigodot$, in place of the future next time modality, $\bigcirc$. Here, however, we present the definitions in a more hierarchic (and logical) fashion. Note that the `Zince` rule defines the past time correspondent to the future time unless, or weak until, modality, i.e. it is a weak version of `Since`.

$$\underline{\text{min}}\ \texttt{Since}(\underline{\text{Form}}\ F_1, \underline{\text{Form}}\ F_2) = F_2 \vee (F_1 \wedge \bigodot \texttt{Since}(F_1, F_2))$$
$$\underline{\text{min}}\ \texttt{EventuallyInPast}(\underline{\text{Form}}\ F) = \texttt{Since}(\underline{\text{true}}, F)$$
$$\underline{\text{max}}\ \texttt{AlwaysInPast}(\underline{\text{Form}}\ F) = \neg\texttt{EventuallyInPast}(\neg F)$$
$$\underline{\text{max}}\ \texttt{Zince}(\underline{\text{Form}}\ F_1, \underline{\text{Form}}\ F_2) = \texttt{Since}(F_1, F_2) \vee \texttt{AlwaysInPast}(F_1)$$

Combined Future and Past Time LTL: By combining the definitions for the future and past time LTLs defined above, we obtain a temporal logic over the future, present and past, in which one can freely intermix the future and past time modalities (to any depth)[2]. We are thus able to express constraints such as if ever the variable x exceeds 0, there was an earlier moment when the variable y was 4 and then remains with that value until it is increased sometime later, possibly after the moment when x exceeds 0.

$$\underline{\text{mon}}\ M_2 = \texttt{Always}(x > 0 \rightarrow \texttt{EventuallyInPast}(y = 4 \wedge \texttt{Until}(y = 4, y > 4)))$$

Extended LTL and μTL: The ability to define temporal modalities recursively provides the ability to define Wolper's ETL or the semantically equivalent fixpoint temporal

[2] See [4] for the correctness argument for such an embedding of propositional LTL in EAGLE.

calculus. Such expressiveness is required to capture regular properties such as temporal formula F is required to be true on every even moment of time:

$$\underline{\max}\ \texttt{Even}(\underline{\text{Form}}\ F) = F \wedge \bigcirc \bigcirc \texttt{Even}(F)$$

The μTL formula $\nu x.(p \wedge \bigcirc \bigcirc x \wedge \mu y.((q \wedge \bigcirc x) \vee \odot y))$, where p and q are atomic formulas, would be denoted by the formula, $X()$, where rules X and Y are:

$$\underline{\max}\ \texttt{X}() = p \wedge \bigcirc \bigcirc \texttt{X}() \wedge \texttt{Y}() \qquad \underline{\min}\ \texttt{Y}() = (q \wedge \bigcirc \texttt{X}()) \vee \odot \texttt{Y}()$$

Extended Regular Expressions: The language of Extended Regular Expressions (ERE), i.e. adding complementation to regular expressions, has been proposed as a powerful formalism for runtime monitoring. EREs can straightforwardly be embedded within our rule-based system. Given, $E ::= \emptyset|\epsilon|a|E \cdot E|E + E|E \cap E|\neg E|E^*$, let $\mathbf{Tr}(E)$ denote the ERE E's corresponding EAGLE formula. For convenience, we define the rule $\underline{\max}\ \texttt{Empty}() = \neg\bigcirc\underline{\text{true}}$ which is true only when evaluated on an empty (suffix) sequence. **Tr** is inductively defined as follows.

$$\begin{array}{lll}
\mathbf{Tr}(\emptyset) & = \underline{\text{false}} & \mathbf{Tr}(\epsilon) = \texttt{Empty}() \\
\mathbf{Tr}(a) & = a \wedge \bigcirc\texttt{Empty}() & \mathbf{Tr}(E_1 \cdot E_2) = \mathbf{Tr}(E_1) \cdot \mathbf{Tr}(E_2) \\
\mathbf{Tr}(E_1 + E_2) & = \mathbf{Tr}(E_1) \vee \mathbf{Tr}(E_2) & \mathbf{Tr}(E_1 \cap E_2) = \mathbf{Tr}(E_1) \wedge \mathbf{Tr}(E_2) \\
\mathbf{Tr}(\neg E) & = \neg\mathbf{Tr}(E) & \\
\mathbf{Tr}(E^*) & = X() \text{ where } \underline{\max}\ X() = \texttt{Empty}() \ \vee\ (\mathbf{Tr}(E) \cdot X()) &
\end{array}$$

Real Time as a Special Case of Data Binding: Metric temporal logics, in which temporal modalities are parameterized by some underlying real-time clock(s), can be straightforwardly embedded into our system through rule parameterization. For example, consider the metric temporal modality, $\diamond^{[t_1,t_2]}$ in a system with just one global clock. An absolute interpretation of $\diamond^{[t_1,t_2]}F$ has the formula true if and only if F holds at some time in the future when the real-time clock has a value within the interval $[t_1, t_2]$. For our purposes, we assume that the states being monitored are time-stamped and that the variable $clock$ holds the value of the real-time clock for the associated state. The rule

$$\begin{array}{l}
\underline{\min}\ \texttt{EventAbs}(\underline{\text{Form}}\ F, \underline{\text{float}}\ t_1, \underline{\text{float}}\ t_2) = \\
\quad (F \wedge t_1 \leq clock \wedge clock \leq t_2) \vee \\
\quad ((clock < t_1 \vee (\neg F \wedge clock \leq t_2)) \wedge \bigcirc\texttt{EventAbs}(F, t_1, t_2))
\end{array}$$

defines the operator $\diamond^{[t_1,t_2]}$ for absolute values of the clock. The rule will succeed when the formula F evaluates to true and $clock$ is within the specified interval $[t_1, t_2]$. If either the formula F doesn't hold and the time-stamp is within the upper time bound, or the lower bound hasn't been reached, then the rule is applied to the next input state. Note that the rule will fail as soon as either the time-stamp is beyond the given interval, i.e. $clock > t_2$, and the formula F has not been satisfied, or the end of the input trace has been passed. A relativized version of the modality can then be defined as:

$$\underline{\min}\ \texttt{EventRel}(\underline{\text{Form}}\ F, \underline{\text{float}}\ t_1, \underline{\text{float}}\ t_2) = \texttt{EventAbs}(F, clock + t_1, clock + t_2)$$

Counting and Statistical Calculations: In a monitoring context, one may wish to gather statistics on the truth of some property, for example whether a particular state property F holds with at least some probability p over a given sequence, i.e. it doesn't fail with probability greater than $(1-p)$. Consider the operator $\Box_p F$ defined by:

$$\sigma, i \models \Box_p F \text{ iff } \exists S \subseteq \{i..|\sigma|\} \text{ s.t. } \frac{|S|}{|\sigma| - i} \geq p \wedge \forall j \in S \,.\, \sigma, j \models F$$

An encoding within our logic can then be given as:

$$\begin{aligned}
&\underline{\text{min}}\ \texttt{A}(\underline{\text{Form}}\ F, \underline{\text{float}}\ p, \underline{\text{int}}\ f, \underline{\text{int}}\ t) = \\
&\quad (\bigcirc\texttt{Empty}() \wedge ((F \wedge (1 - \tfrac{f}{t}) >= p) \vee (\neg F \wedge (1 - \tfrac{f+1}{t}) >= p))) \vee \\
&\quad (\neg\texttt{Empty}() \wedge ((F \rightarrow \bigcirc\texttt{A}(F, p, f, t+1)) \wedge (\neg F \rightarrow \bigcirc\texttt{A}(F, p, f+1, t+1))))
\end{aligned}$$

$$\underline{\text{min}}\ \texttt{AtLeast}(\underline{\text{Form}}\ F, \underline{\text{float}}\ p) = \texttt{A}(F, p, 0, 1)$$

The auxiliary rule `A` counts the number of failures of F in its argument f and the number of events monitored in argument t. Thus, at the end of monitoring, the first line of `A`'s body determines whether F has held with the desired probability, i.e. $\geq p$. `AtLeast` therefore calls `A` with arguments f and t initialized to 0 and 1 respectively.

Towards Context Free: Above we showed that EAGLE could encode logics such as ETL, which extend LTL with regular grammars (when restricted to finite traces), or even extended regular expressions. In fact, we can go beyond regularity into the world of context-free languages, necessary, for example, to express properties such as every login is matched by a logout and at no point are there more logouts than logins. Indeed, such a property can be expressed in several ways in EAGLE. Assume we are monitoring a sequence of login and logout events, characterized, respectively, by the formulas $login$ and $logout$. We can define a rule $\texttt{Match}(\underline{\text{Form}}\ F_1, \underline{\text{Form}}\ F_2)$ and monitor with $\texttt{Match}(login, logout)$ where:

$$\underline{\text{min}}\ \texttt{Match}(\underline{\text{Form}}\ F_1, \underline{\text{Form}}\ F_2) = F_1 \cdot \texttt{Match}(F_1, F_2) \cdot F_2 \cdot \texttt{Match}(F_1, F_2) \ \vee\ \texttt{Empty}()$$

Less elegantly, and which we leave as an exercise, one could use the rule parametrization mechanism to count the numbers of logins and logouts.

3 Algorithm

In this section, we briefly outline the computation mechanism used to determine whether a given monitoring formula holds for some given input sequence of events. For details on the algorithm, the interested readers can refer to [5]. In the algorithm, we assume that a local state is maintained on the observer side. The *expressions* or *propositions* are specified with respect to the variables in this local state. At every event the observer modifies the local state of the observer, based on that event, and then evaluates the monitored formulas on the new state, and generates a new set of monitored formulas. At the end of the trace the values of the monitored formulas are determined. If the value of a formula is true, the formula is satisfied, otherwise the formula is violated.

First, a monitor formula F is transformed to another formula F'. This transformation addresses the semantics of EAGLE at the beginning of a trace. Next, the transformed formula is monitored against an execution trace by repeated application of *eval*. The evaluation of a formula F on a state $s = \sigma(i)$ in a trace σ results in an another formula $\mathit{eval}\langle\!\langle F, s\rangle\!\rangle$ with the property that $\sigma, i \models F$ if and only if $\sigma, i+1 \models \mathit{eval}\langle\!\langle F, s\rangle\!\rangle$. The definition of the function $\mathit{eval} : \underline{\text{Form}} \times \underline{\text{State}} \rightarrow \underline{\text{Form}}$ uses an auxiliary function *update* with signature $\mathit{update} : \underline{\text{Form}} \times \underline{\text{State}} \rightarrow \underline{\text{Form}}$. *update*'s role is to pre-evaluate a formula if it is guarded by the previous operator $\odot$. Formally, *update* has the property that $\sigma, i \models \bigcirc F$ iff $\sigma, i+1 \models \mathit{update}\langle\!\langle F, s\rangle\!\rangle$. Had there been no past time modality in EAGLE *update* would be unnecessary and the identity $\sigma, i \models \bigcirc F$ iff $\sigma, i+1 \models F$ could have been used. At the end (or at the beginning) of a trace, the function $\mathit{value} : \mathit{Form} \rightarrow \{\underline{\text{true}}, \underline{\text{false}}\}$ when applied on F returns $\underline{\text{true}}$ iff $\sigma, |\sigma| + 1 \models F$ (or $\sigma, 0 \models F$) and returns $\underline{\text{false}}$ otherwise. Thus given a sequence of states $s_1 s_2 \ldots s_n$, an EAGLE formula F is said to be satisfied by the sequence of states if and only if $\mathit{value}\langle\!\langle \mathit{eval}\langle\!\langle \ldots \mathit{eval}\langle\!\langle \mathit{eval}\langle\!\langle F', s_1\rangle\!\rangle, s_2\rangle\!\rangle \ldots, s_n\rangle\!\rangle\rangle\!\rangle$ is $\underline{\text{true}}$. The functions *eval*, *update* and *value* are the basis of the calculus for our rule-based framework.

3.1 Calculus

The *eval*, *update* and *value* functions are defined a priori for all operators except for the rule application. The definitions of *eval*, *update* and *value* for rules get generated based on the definition of rules in the specification. The definitions of *eval*, *update* and *value* on the different primitive operators are given below.

$$
\begin{aligned}
\mathit{eval}\langle\!\langle \underline{\text{true}}, s\rangle\!\rangle &= \underline{\text{true}} \\
\mathit{eval}\langle\!\langle \underline{\text{false}}, s\rangle\!\rangle &= \underline{\text{false}} \\
\mathit{eval}\langle\!\langle \mathit{jexp}, s\rangle\!\rangle &= \text{ value of } \mathit{jexp} \text{ in } s \\
\mathit{eval}\langle\!\langle F_1 \mathit{\ op\ } F_2, s\rangle\!\rangle &= \mathit{eval}\langle\!\langle F_1, s\rangle\!\rangle \mathit{\ op\ } \mathit{eval}\langle\!\langle F_2, s\rangle\!\rangle \\
\mathit{eval}\langle\!\langle \neg F, s\rangle\!\rangle &= \neg \mathit{eval}\langle\!\langle F, s\rangle\!\rangle \\
\mathit{eval}\langle\!\langle \bigcirc F, s\rangle\!\rangle &= \mathit{update}\langle\!\langle F, s\rangle\!\rangle \\
\mathit{eval}\langle\!\langle F_1 \cdot F_2, s\rangle\!\rangle &= \\
&= \begin{cases} \mathit{eval}\langle\!\langle F_1, s\rangle\!\rangle \cdot F_2 \text{ if } \mathit{value}\langle\!\langle F_1\rangle\!\rangle = \underline{\text{false}} \\ (\mathit{eval}\langle\!\langle F_1, s\rangle\!\rangle \cdot F_2) \vee \mathit{eval}\langle\!\langle F_2, s\rangle\!\rangle \text{ otherwise} \end{cases}
\end{aligned}
\qquad
\begin{aligned}
\mathit{value}\langle\!\langle \underline{\text{true}}\rangle\!\rangle &= \underline{\text{true}} \\
\mathit{value}\langle\!\langle \underline{\text{false}}\rangle\!\rangle &= \underline{\text{false}} \\
\mathit{value}\langle\!\langle \mathit{jexp}\rangle\!\rangle &= \underline{\text{false}} \\
\mathit{value}\langle\!\langle F_1 \mathit{\ op\ } F_2\rangle\!\rangle &= \mathit{value}\langle\!\langle F_1\rangle\!\rangle \mathit{\ op\ } \mathit{value}\langle\!\langle F_2\rangle\!\rangle \\
\mathit{value}\langle\!\langle \neg F\rangle\!\rangle &= \neg \mathit{value}\langle\!\langle F\rangle\!\rangle \\
\mathit{value}\langle\!\langle \bigcirc F\rangle\!\rangle & \\
&= \begin{cases} F & \text{if at the beginning of trace} \\ \underline{\text{false}} & \text{if at the end of trace} \end{cases} \\
\mathit{value}\langle\!\langle F_1 \cdot F_2\rangle\!\rangle &= \mathit{value}\langle\!\langle F_1\rangle\!\rangle \wedge \mathit{value}\langle\!\langle F_2\rangle\!\rangle
\end{aligned}
$$

$$
\begin{aligned}
\mathit{update}\langle\!\langle \underline{\text{true}}, s\rangle\!\rangle &= \underline{\text{true}} \\
\mathit{update}\langle\!\langle \underline{\text{false}}, s\rangle\!\rangle &= \underline{\text{false}} \\
\mathit{update}\langle\!\langle \mathit{jexp}, s\rangle\!\rangle &= \mathit{jexp} \\
\mathit{update}\langle\!\langle F_1 \mathit{\ op\ } F_2, s\rangle\!\rangle &= \mathit{update}\langle\!\langle F_1, s\rangle\!\rangle \mathit{\ op\ } \mathit{update}\langle\!\langle F_2, s\rangle\!\rangle \\
\mathit{update}\langle\!\langle \neg F, s\rangle\!\rangle &= \neg \mathit{update}\langle\!\langle F, s\rangle\!\rangle \\
\mathit{update}\langle\!\langle F_1 \cdot F_2, s\rangle\!\rangle &= \mathit{update}\langle\!\langle F_1, s\rangle\!\rangle \cdot F_2
\end{aligned}
$$

In the above definitions, *op* can be $\wedge, \vee, \rightarrow$. In most of the definitions we simply propagate the function to the subformulas. However, the concatenation operator is handled in a special way. The *eval* of a formula $F_1 \cdot F_2$ on a state s first checks if $\mathit{value}\langle\!\langle F_1\rangle\!\rangle$ is true or not. If the value is true then one can *non-deterministically* split the trace just before the state s. Hence, the evaluation becomes $(\mathit{eval}\langle\!\langle F_1, s\rangle\!\rangle \cdot F_2) \vee \mathit{eval}\langle\!\langle F_2, s\rangle\!\rangle$ where $\vee$ expresses the non-determinism. Otherwise, if the trace cannot be split, the evaluation becomes simply $\mathit{eval}\langle\!\langle F_1, s\rangle\!\rangle \cdot F_2$. The function *update* on the formula $F_1 \cdot F_2$ simply

updates the formula F_1, as F_2 is not effected by the trace that effects F_1. At the end of a trace, that $F_1 \cdot F_2$ is satisfied means that the remaining empty trace can be split into two empty traces satisfying respectively F_1 and F_2; hence the conjunction in $\mathit{value}\langle\!\langle F_1 \cdot F_2 \rangle\!\rangle$. Since the semantics of $\bigcirc$ is different at the beginning and at the end of a trace, we have to consider the two cases in the definition of *value* for $\bigcirc F$.

The operator $\odot$ requires special attention. If a formula F is guarded by a previous operator then we evaluate F at every event and use the result of this evaluation in the next state. Thus, the result of evaluating F is required to be stored in some temporary placeholder so that it can be used in the next state. To allocate a placeholder for a $\odot$ operator, we introduce the operator $\underline{\text{Previous}} : \underline{\text{Form}} \times \underline{\text{Form}} \rightarrow \underline{\text{Form}}$. The second argument for this operator acts as the placeholder. We transform a formula $\odot F$ at the beginning of monitoring as follows:

$$\odot F \rightarrow \underline{\text{Previous}}(F', \mathit{value}\langle\!\langle F' \rangle\!\rangle) \text{ where } F' \text{ is the transformed version of } F$$

We define *eval*, *update*, and *value* for $\underline{\text{Previous}}$ as follows:

$$\begin{aligned}
&\mathit{eval}\langle\!\langle \underline{\text{Previous}}(F, \mathit{past}), s \rangle\!\rangle = \mathit{eval}\langle\!\langle \mathit{past}, s \rangle\!\rangle \\
&\mathit{update}\langle\!\langle \underline{\text{Previous}}(F, \mathit{past}), s \rangle\!\rangle = \underline{\text{Previous}}(\mathit{update}\langle\!\langle F, s \rangle\!\rangle, \mathit{eval}\langle\!\langle F, s \rangle\!\rangle) \\
&\mathit{value}\langle\!\langle \underline{\text{Previous}}(F, \mathit{past}) \rangle\!\rangle = \begin{cases} \underline{\text{false}} & \text{if at the beginning of trace} \\ \mathit{value}\langle\!\langle \mathit{past} \rangle\!\rangle & \text{if at the end of trace} \end{cases}
\end{aligned}$$

Here, *eval* of $\underline{\text{Previous}}(F, \mathit{past})$ returns the *eval* of the second argument of $\underline{\text{Previous}}$, *past*, that contains the evaluation of F in the previous state. In *update* we not only update the first argument F but also evaluate F and pass it as the second argument of $\underline{\text{Previous}}$. Thus in the next state the second argument of $\underline{\text{Previous}}$, *past*, is bound to $\odot F$. The $\mathit{value}\langle\!\langle F' \rangle\!\rangle$ that appears in the transformation of $\odot F$ is the value of F' at the beginning of the trace. This takes care of the semantics of EAGLE at the beginning of a trace.

3.2 Monitor Synthesis for Rules

In what follows, $\rho b.H(b)$ denotes a recursive structure where free occurrences of b in H point back to $\rho b.H(b)$. Formally, $\rho b.H(b)$ is a closed form term that denotes a fix-point solution to the equation $x = H(x)$ and hence $\rho b.H(b) = H(\rho b.H(b))$. The open form $H(b)$ denotes a formula with free recursion variable b. In structural terms, a solution to $x = H(x)$ can be represented as a graph structure where the leaves, denoted by x, point back to the root node of the graph. Our implementation uses this structural solution.

We replace every rule R by an operator $\underline{\text{R}}$ during transformation. For a subformula $\texttt{R}(\overline{F}, \overline{P})$, where the rule R is defined as $\{\underline{\text{max}}|\underline{\text{min}}\}\ \texttt{R}(\overline{\underline{\text{Form}}}\ \overline{f}, \overline{T}\ \overline{p}) = B$, we transform the subformula as follows:

$$\begin{aligned}
&\texttt{R}(\overline{F}, \overline{P}) \rightarrow \underline{\text{R}}(\rho b.B'[\overline{f} \mapsto \overline{F'}], \overline{P}) \\
&\qquad \text{where } B' \text{ and } F' \text{ are transformed versions of } B \text{ and } F \text{ respectively} \\
&\texttt{R}(\overline{F}, \overline{P'}) \rightarrow \underline{\text{R}}(b, \overline{P'}) \text{ where } \texttt{R}(\overline{F}, \overline{P'}) \text{ is a subformula of } B
\end{aligned}$$

The second equation is invoked if a recursion is detected [3]; that is while transforming B if $\texttt{R}(\overline{F}, \overline{P'})$ is encountered as a subformula of B. Note that here the variable b should

[3] A formal description of recursion detection mechanism can be found in [5].

be a fresh name to avoid possible variable capturing. For example consider the formula $\Box(x > 0 \rightarrow \exists k(k = x \wedge \Diamond(z > 0 \wedge y = k)))$. A specification for this monitor can be presented in EAGLE as follows:

$$\begin{array}{l}\underline{\max}\ \mathtt{A}(\underline{\mathrm{Form}}\ f) = f \wedge \bigcirc \mathtt{A}(f)\\ \underline{\min}\ \mathtt{Ep}(\underline{\mathrm{Form}}\ f) = f \vee \odot \mathtt{Ep}(f)\\ \underline{\min}\ \mathtt{Ev}(\underline{\mathrm{int}}\ k) = \mathtt{Ep}(z > 0 \wedge y = k)\\ \underline{\mathrm{mon}}\ M = \mathtt{A}(x > 0 \rightarrow \mathtt{Ev}(x))\end{array}$$

The transformed version of M is as follows:

$$\begin{array}{l}M' = \underline{\mathrm{A}}(\rho b_1.((x > 0) \rightarrow\\ \qquad \underline{\mathrm{Ev}}(\rho b_2.\underline{\mathrm{Ep}}(\rho b_3.((z > 0) \wedge (y = k) \vee \underline{\mathrm{Previous}}(\underline{\mathrm{Ep}}(b_3), \underline{\mathrm{false}}))), x)) \wedge \underline{\mathrm{Next}}(\underline{\mathrm{A}}(b_1)))\end{array}$$

The definitions of *update*, *eval* and *value* for $\underline{\mathrm{R}}$ are as follows:

$$\begin{array}{l}\mathit{update}\langle\!\langle \underline{\mathrm{R}}(\rho b.H(b), \overline{P}), s\rangle\!\rangle = \underline{\mathrm{R}}(\rho b'.\mathit{update}\langle\!\langle H(\rho b.H(b)), s\rangle\!\rangle, \overline{P})\\ \mathit{update}\langle\!\langle \underline{\mathrm{R}}(\rho b.H(b), \overline{P}), s\rangle\!\rangle = \underline{\mathrm{R}}(b', \overline{P})\\ \qquad\qquad\qquad \text{if } \underline{\mathrm{R}}(\rho b.H(b), \overline{P}) \text{ is a subformula of } H(\rho b.H(b))\end{array}$$

Here, $\rho b.H(b)$ is first expanded to $H(\rho b.H(b))$ and then *update* is applied on it; that is, the body of the rule is updated. The second equation detects a recursion, that is, update of $H(\rho b.H(b))$ encounters $\underline{\mathrm{R}}(\rho b.H(b), \overline{P})$ as a subformula of $H(\rho b.H(b))$. In that case $\underline{\mathrm{R}}(b', \overline{F}, \overline{P})$ is returned terminating the recursion.

$$\mathit{eval}\langle\!\langle \underline{\mathrm{R}}(\rho b.H(b), \overline{P}), s\rangle\!\rangle = \mathit{eval}\langle\!\langle H(\rho b.H(b))[\overline{p} \mapsto \mathit{eval}\langle\!\langle \overline{P}, s\rangle\!\rangle], s\rangle\!\rangle$$

Here, $\rho b.H(b)$ is first expanded to $H(\rho b.H(b))$ and then any arguments of primitive type are evaluated and substituted in the expansion. The function *eval* is then applied on the expansion. Note that the result of $\mathit{eval}\langle\!\langle P, s\rangle\!\rangle$, where P is an expression, may be a partially evaluated expression if expressions referred to by some of the variables in P are partially evaluated. The expression gets fully evaluated once all the variables referred to by the expressions are fully evaluated.

$$\mathit{value}\langle\!\langle \underline{\mathrm{R}}(B, \overline{P})\rangle\!\rangle = \underline{\mathrm{false}} \text{ if } \mathtt{R} \text{ is minimal} \qquad \mathit{value}\langle\!\langle \underline{\mathrm{R}}(B, \overline{P})\rangle\!\rangle = \underline{\mathrm{true}} \text{ if } \mathtt{R} \text{ is maximal}$$

The *value* of a $\underline{\max}$ rule is $\underline{\mathrm{true}}$ and that of a $\underline{\min}$ rule is $\underline{\mathrm{false}}$.

For example, for a sequence of states sequence $\{x = 0, y = 3, z = 1\}, \{x = 0, y = 5, z = 2\}, \{x = 2, y = 2, z = 0\}$, step-by-step monitoring of the formula M' on this sequence takes place as follows:

Step 1: $s = \{x = 0, y = 3, z = 1\}$

$$\begin{array}{l}F_1 = \mathit{eval}\langle\!\langle F, s\rangle\!\rangle\\ \quad = \underline{\mathrm{A}}(\rho b_1.((x > 0) \rightarrow\\ \qquad \underline{\mathrm{Ev}}(\rho b_2.\underline{\mathrm{Ep}}(\rho b_3.((z > 0) \wedge (y = k) \vee \underline{\mathrm{Previous}}(\underline{\mathrm{Ep}}(b_3), (3 = k)))), x)) \wedge \underline{\mathrm{Next}}(\underline{\mathrm{A}}(b_1)))\end{array}$$

Observe that in the above step the second argument of $\underline{\mathrm{Previous}}$ is partially evaluated as the value of k is not available. The value of k becomes available when we apply *eval* on $\underline{\mathrm{Ev}}$. At that time *eval* also replaces the free variable k appearing in the first argument of $\underline{\mathrm{Ev}}$ by the actual value. This replacement can easily be seen if the definition of *eval* of $\underline{\mathrm{Ev}}$ is written down.

Step 2: $s = \{x = 0, y = 5, z = 2\}$

$$\begin{aligned} F_2 &= \mathit{eval}\langle\!\langle F_1, s\rangle\!\rangle \\ &= \underline{\mathrm{A}}(\rho b_1.((x > 0) \rightarrow \underline{\mathrm{Ev}}(\rho b_2.\underline{\mathrm{Ep}}(\rho b_3.((z > 0) \wedge (y = k) \vee \\ &\qquad \underline{\mathrm{Previous}}(\underline{\mathrm{Ep}}(b_3), (3 = k) \vee (5 = k)))), x)) \wedge \underline{\mathrm{Next}}(\underline{\mathrm{A}}(b_1))) \end{aligned}$$

Step 3: $s = \{x = 2, y = 2, z = 0\}$

$$F_3 = \mathit{eval}\langle\!\langle F_2, s\rangle\!\rangle = \underline{\mathrm{false}}$$

Thus the formula is violated on the third state of the trace.

4 Implementation and Experiments

We have implemented this monitoring framework in Java. The implemented system works in two phases. First, it compiles the specification file to generate a set of Java classes; a class is generated for each rule. Second, the Java class files are compiled into Java bytecode and then the monitoring engine runs on a trace; the engine dynamically loads the Java classes for rules at monitoring time.

Our implementation of propositional logic uses the decision procedure of Hsiang [15]. The procedure reduces a tautological formula to the constant true, a false formula to the constant false, and all other formulas to canonical forms which are exclusive or ($\oplus$) of conjunctions. The procedure is given below using equations that are shown to be Church-Rosser and terminating modulo associativity and commutativity.

$\mathrm{true} \wedge \phi = \phi$	$\mathrm{false} \wedge \phi = \mathrm{false}$	$\phi_1 \wedge (\phi_2 \oplus \phi_3) = (\phi_1 \wedge \phi_2) \oplus (\phi_1 \wedge \phi_3)$
$\phi \wedge \phi = \phi$	$\mathrm{false} \oplus \phi = \phi$	$\phi_1 \vee \phi_2 = (\phi_1 \wedge \phi_2) \oplus \phi_1 \oplus \phi_2$
$\phi \oplus \phi = \mathrm{false}$	$\neg\phi = \mathrm{true} \oplus \phi$	$\phi_1 \rightarrow \phi_2 = \mathrm{true} \oplus \phi_1 \oplus (\phi_1 \wedge \phi_2)$
		$\phi_1 \equiv \phi_2 = \mathrm{true} \oplus \phi_1 \oplus \phi_2$

The above equations ensure that the size of a formula is small. In the translational phase, a Java class is generated for each rule in the specification. The Java class contains a constructor, a `value` method, an `eval` method, and a `update` method corresponding to the *value*, *eval* and *update* operators in the calculus. The arguments are made fields in the class and they are initialized through the constructor. The choice of generating Java classes for the rules was made in order to achieve an efficient implementation. To handle partial evaluation we wrap every Java expression in a Java class. Each of those classes contains a method `isAvailable()` that returns `true` whenever the Java expression representing that class is fully evaluated and returns `false` otherwise. The class also stores, as fields, the different other Java expression objects corresponding to the different variables (formula variables and state variables) that it uses in its Java expression. Once all those Java expressions are fully evaluated, the object for the Java expression evaluates itself and any subsequent call of `isAvailable()` on this object returns `true`.

Once all the Java classes have been generated, the engine compiles all the generated Java classes, creates a list of monitors (which are also formulas) and starts monitoring all of them. During monitoring the engine takes the states from the trace, one by one, and evaluates the list of monitors on each to generate another list of formulas that become the new monitors for the next state. If at any point a monitor (a formula) becomes false an error message is generated and that monitor is removed from the list. At the end of

a trace the value of each monitor is calculated and if false, a warning message for the particular monitor is generated. The details of the implementation are beyond the scope of the paper. However, interested readers can get the tool from the authors.

EAGLE has been applied to test a planetary rover controller in a collaborative effort with other colleagues, see [2] for an earlier similar experiment using a simpler logic. The rover controller, written in 35,000 lines of C++, executes action plans. The testing environment, consists of a test-case generator, automatically generating input plans for the controller. Additionally, for each input plan a set of temporal formulas is generated that the plan execution should satisfy. The controller is executed on the generated plans and the implementation of EAGLE is used to monitor that execution traces satisfy the formulas. The automated testing system found a missing feature that had been overlooked by the developers: the lower bounds on action execution duration were not checked by the implementation, causing some executions to succeed while they in fact should fail due to the too early termination of some action execution. The temporal formulas, however, correctly predicted failure in these cases. This error showed up later during actual rover operation before it was corrected.

5 Conclusion and Future Work

We have presented the succinct and powerful logic EAGLE, based on recursive parameterized rule definitions over three primitive temporal operators. We have indicated its power by expressing some other sophisticated logics in it. Initial experiments have been successful. Future work includes: optimizing the current implementation; supporting user-defined surface syntax; associating actions with formulas; and incorporating automated program instrumentation.

References

1. *1st, 2nd and 3rd CAV Workshops on Runtime Verification (RV'01 - RV'03)*, volume 55(2), 70(4), 89(2) of *ENTCS*. Elsevier Science: 2001, 2002, 2003.
2. C. Artho, D. Drusinsky, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, G. Roşu, and W. Visser. Experiments with Test Case Generation and Runtime Analysis. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines (ASM'03)*, volume 2589 of *LNCS*, pages 87–107. Springer, March 2003.
3. H. Barringer, M. Fisher, D. Gabbay, G. Gough, and R. Owens. METATEM: An Introduction. *Formal Aspects of Computing*, 7(5):533–549, 1995.
4. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Eagle does Space Efficient LTL Monitoring. Pre-Print CSPP-25, University of Manchester, Department of Computer Science, October 2003. Download: http://www.cs.man.ac.uk/cspreprints/PrePrints/cspp25.pdf.
5. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Eagle Monitors by Collecting Facts and Generating Obligations. Pre-Print CSPP-26, University of Manchester, Department of Computer Science, October 2003. Download: http://www.cs.man.ac.uk/cspreprints/PrePrints/cspp26.pdf.
6. D. Drusinsky. The Temporal Rover and the ATG Rover. In K. Havelund, J. Penix, and W. Visser, editors, *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 323–330. Springer, 2000.

7. D. Drusinsky. Monitoring Temporal Rules Combined with Time Series. In *CAV'03*, volume 2725 of *LNCS*, pages 114–118. Springer-Verlag, 2003.
8. B. Finkbeiner, S. Sankaranarayanan, and H. Sipma. Collecting Statistics over Runtime Executions. In *Proceedings of the 2nd International Workshop on Runtime Verification (RV'02)* [1], pages 36–55.
9. B. Finkbeiner and H. Sipma. Checking Finite Traces using Alternating Automata. In *Proceedings of the 1st International Workshop on Runtime Verification (RV'01)* [1], pages 44–60.
10. D. Gabbay. The Declarative Past and Imperative Future: Executable Temporal Logic for Interactive Systems. In *Proceedings of the 1st Conference on Temporal Logic in Specification, Altrincham, April 1987*, volume 398 of *LNCS*, pages 409–448, 1989.
11. D. Giannakopoulou and K. Havelund. Automata-Based Verification of Temporal Properties on Running Programs. In *Proceedings, International Conference on Automated Software Engineering (ASE'01)*, pages 412–416. ENTCS, 2001. Coronado Island, California.
12. K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. In *Proceedings of the 1st International Workshop on Runtime Verification (RV'01)* [1], pages 97–114. Extended version to appear in the journal: Formal Methods in System Design, Kluwer, 2004.
13. K. Havelund and G. Roşu. Monitoring Programs using Rewriting. In *Proceedings, International Conference on Automated Software Engineering (ASE'01)*, pages 135–143. Institute of Electrical and Electronics Engineers, 2001. Coronado Island, California.
14. K. Havelund and G. Roşu. Synthesizing Monitors for Safety Properties. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*, pages 342–356. Springer, 2002. Extended version to appear in the journal: Software Tools for Technology Transfer, Springer, 2004.
15. Jieh Hsiang. Refutational Theorem Proving using Term Rewriting Systems. *Artificial Intelligence*, 25:255–300, 1985.
16. D. Kortenkamp, T. Milam, R. Simmons, and J. Fernandez. Collecting and Analyzing Data from Distributed Control Programs. In *Proceedings of the 1st International Workshop on Runtime Verification (RV'01)* [1], pages 133–151.
17. K. Jelling Kristoffersen, C. Pedersen, and H. R. Andersen. Runtime Verification of Timed LTL using Disjunctive Normalized Equation Systems. In *Proceedings of the 3rd International Workshop on Runtime Verification (RV'03)* [1], pages 146–161.
18. I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime Assurance Based on Formal Specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.
19. A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–77, 1977.
20. K. Sen and G. Roşu. Generating Optimal Monitors for Extended Regular Expressions. In *Proceedings of the 3rd International Workshop on Runtime Verification (RV'03)* [1], pages 162–181.

On the Expressive Power of Canonical Abstraction*

Mooly Sagiv

School of Computer Science
Tel-Aviv University
Tel-Aviv 69978
Israel msagiv@acm.org

Abstract. Abstraction and abstract interpretation are key tools for automatically verifying properties of systems. One of the major challenges in abstract interpretation is how to obtain abstractions that are precise enough to provide useful information.
In this talk, I will survey a parametric abstract domain called *canonical abstraction* which was motivated by the shape analysis problem of determining "shape invariants" for programs that perform destructive updating on dynamically allocated storage. The shape analysis problem was originally defined by [1]. Canonical abstraction was originally defined in [2]. A system for abstract interpretation based on abstract interpretation was defined in [3,4].
I will discuss properties that have been verified using this abstraction. A couple of interesting properties of this abstract domain will be presented. Finally, I will also show some of the limitations of this domain.

References

1. N.D. Jones and S.S. Muchnick, Flow Analysis and Optimization of Lisp-Like Structures, *Program Flow Analysis: Theory and Applications*, Prentice Hall, pp. 102–131, 1981.
2. M. Sagiv and T. Reps and R. Wilhelm, Parametric Shape Analysis via 3-Valued Logic, *ACM Transactions on Programming Languages and Systems*, Vol. 23, No. 3, pp. 217–298, 2002
3. T. Lev-Ami and M. Sagiv, TVLA: A System for Implementing Static Analyses, *SAS*, Springer, LNCS 1824, pp. 280–301, 2000.
4. R. Manevich, G. Ramalingam, J. Field, D.Goyal, and M. Sagiv, Compactly Representing First-Order Structures for Static Analysis, *SAS* , Springer, LNCS 2477, pp. 196–212 .

* This is a joint work with Thomas Reps and Reinhard Wilhelm.

B. Steffen and G. Levi (Eds.): VMCAI 2004, LNCS 2937, p. 58, 2004.

Boolean Algebra of Shape Analysis Constraints*

Viktor Kuncak and Martin Rinard

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139, USA
{vkuncak,rinard}@csail.mit.edu

Abstract. The parametric shape analysis framework of Sagiv, Reps, and Wilhelm [45,46] uses three-valued structures as dataflow lattice elements to represent sets of states at different program points. The recent work of Yorsh, Reps, Sagiv, Wilhelm [48,50] introduces a family of formulas in (classical, two-valued) logic that are isomorphic to three-valued structures [46] and represent the same sets of concrete states.
In this paper we introduce a larger syntactic class of formulas that has the same expressive power as the formulas in [48]. The formulas in [48] can be viewed as a normal form of the formulas in our syntactic class; we give an algorithm for transforming our formulas to this normal form. Our formulas make it obvious that the constraints are closed under all boolean operations and therefore form a boolean algebra. Our algorithm also gives a reduction of the entailment and the equivalence problems for these constraints to the satisfiability problem.

Keywords: Shape Analysis, Program Verification, Abstract Interpretation, Boolean Algebra, First-Order Logic, Model Checking

1 Introduction

Background. Shape analysis [46,32,22,20,12,16,15,9,37,27] is a technique for statically analyzing programs that manipulate dynamically allocated data structures, and is important for precise reasoning about programs written in modern imperative programming languages. Parametric shape analysis [45,46] is a framework that can be instantiated to provide a variety of precise shape analyses. We can describe this approach informally as follows. The concrete program state is a *two-valued structure*, that is, a finite relational structure $\langle U^\sharp, \iota^\sharp \rangle$, which maps, for example, a binary relation symbol r to a binary relation $\iota^\sharp(r) : U^\sharp \times U^\sharp \to \{0,1\}$. To represent a potentially infinite set of concrete program states, [46] uses finite *three-valued structures*, which are relational structures in three-valued logic [24, 38]. A three-valued structure $\langle U, \iota \rangle$ maps a binary relation symbol r to a three-valued relation $\iota(r) : U \times U \to \{\{0\},\{1\},\{0,1\}\}$. Three-valued structures generalize the graphs used in several previous shape analyses [44, 22, 9].

* This research was supported in part by DARPA Contract F33615-00-C-1692, NSF Grant CCR00-86154, NSF Grant CCR00-63513, and the Singapore-MIT Alliance.

B. Steffen and G. Levi (Eds.): VMCAI 2004, LNCS 2937, pp. 59–72, 2004.

The elements of the domain U of a three-valued structure $\langle U, \iota \rangle$ represent disjoint non-empty sets of objects. Given two such sets A and B, we can compute the three-valued relation by $\iota(r)(A, B) = \{\iota^\sharp(r)(a, b) \mid a \in A \wedge b \in B\}$. As observed in [48, 50], the fact $\iota(r)(A, B) = \{0\}$ means that the formula $\neg\exists x \exists y. A(x) \wedge B(y) \wedge r(x, y)$ holds on the two-valued structure $\langle U^\sharp, \iota^\sharp \rangle$. Similarly, the fact $\iota(r)(A, B) = \{1\}$ means that $\neg\exists x \exists y. A(x) \wedge B(y) \wedge \neg r(x, y)$ holds, whereas $\iota(r)(A, B) = \{0, 1\}$ means that both $\exists x \exists y. A(x) \wedge B(y) \wedge r(x, y)$ and $\exists x \exists y. A(x) \wedge B(y) \wedge \neg r(x, y)$ hold. As a result, any three-valued structure can be described by a corresponding formula in first-order logic [50]. In this paper we take a closer look at the class of formulas that arise when characterizing the meaning of three-valued structures. We characterize such formulas as the set of all boolean combinations of certain simple formulas, such as $\exists x \exists y. A(x) \wedge B(y) \wedge r(x, y)$ (see Definition 5). As a result, we establish that the meaning of three valued structures (under the tight concretization semantics [50, Chapter 7]) is closed under all boolean operations and therefore forms a boolean algebra.

Characterizing structures using formulas. The characterization of three-valued structures using formulas in first-order logic is presented for the first time in [48, 50]. Section 3.1 of [48] explains that the semantics of general three-valued structures can represent the existence of graph coloring. As a result, first-order structures in general are not definable using first-order logic, but require the use of monadic second-order logic [48, Section 4]. However, an interesting class of three-valued structures can be represented using first-order logic [48, Section 3.2], in particular, this is the case for *bounded structures*. Two versions of the semantics for three-valued structures are of interest: the *standard concretization* [45, Definition 3.5], [48, Chapter 3] and the *tight concretization* [48, Chapter 7] (the later corresponding to the *canonical abstraction* [45, Definition 3.6]). One can view the *characteristic formulas for canonical abstraction* of [48, Chapter 7] as the starting point for the class of formulas in this paper: we show how to allow a richer syntactic class of formulas, and give an algorithm for converting these formulas to the characteristic formulas for canonical abstraction.

We have previously studied *regular graph constraints* [29, 28], inspired by the semantics of role analysis [25, 27, 26]. Regular graph constraints abstract the notion of graph summaries where nodes do not have a unique abstraction criterion. In [28, 29] we observe that such constraints can be equivalently characterized using graphs summaries and using existential monadic second-order logic formulas. Somewhat surprisingly, whereas the satisfiability of regular graph constraints is decidable [29, Section 2.4], the entailment and the equivalence of regular graph constraints are undecidable [29, Section 3], [28]. These properties of regular graph constraints are in contrast to the nice closure properties of the *boolean shape analysis constraints* of the present paper.

1.1 Contributions

Main result. The main result of this paper is a new syntactic class of formulas that characterize the meaning of three-valued structures under tight concretiza-

tion. The new syntactic class is defined as the set of all boolean combinations of formulas of a certain form. The proof of the equivalence of the new syntactic class and previously introduced characteristic formulas for canonical abstraction [48] is a normalization algorithm that transforms formulas in our syntactic class to the characteristic formulas for canonical abstraction (which are isomorphic to three-valued structures). Our characterization immediately implies that the constraints expressible as the meaning of three-valued structures are closed under all boolean operations, we thus call them "boolean shape analysis constraints".

Consequences of boolean closure. The resulting closure properties of boolean shape analysis constraints have several potential uses. The closure under disjunction is necessary for fixpoint computations in dataflow analysis and can easily be computed even for three-valued structures (by taking the union of sets of three-valued structures). What our results show is that boolean shape analysis constraints are also closed under conjunction and negation.

The conjunction of constraints is needed, for example, in compositional interprocedural shape analysis, which computes the relation composition of relations on states. Conjunction allows the analysis to simultaneously retain the call-site specific information that the callee preserves across the call, and the postcondition which summarizes the actions of the callee.

The negation of constraints is useful for expressing deterministic branches in control-flow graphs. For example, an `if` statement with the condition c results in conjoining the dataflow fact d to yield $d \wedge c$ in the `then` branch, and $d \wedge \neg c$ in the `else` branch. Similarly, the `assert`(c) statement, which is an important mechanism for program specification, has (in the relational semantics) the condition $\neg c$ for the branch which leads to an error state.

Finally, the closure under negation implies that both the implication and the equivalence of shape analysis constraints are reducible to the satisfiability of shape analysis constraints. The implication problem is important in compositional shape analysis which uses assume/guarantee reasoning to show that a procedure conforms to its specification.

Decidability of constraints. The closure of boolean shape analysis constraints under boolean operations holds in the presence of arbitrary instrumentation predicates [46, Section 5]. What the particular choice of instrumentation predicates determines is whether the satisfiability problem for the constraints is decidable. If the satisfiability problem for three-valued structures with a particular choice of instrumentation predicates is decidable, our normalization algorithm yields an algorithm for the satisfiability problem of formulas in the richer syntactic class, which, by closure under boolean operations, gives an algorithm for deciding the entailment and the equivalence of boolean shape analysis constraints.

Consequences for program annotations. The ability to write program annotations can greatly improve the effectiveness of static analysis, but the representation of program properties in the program analysis is often different from the representation of program properties that is appropriate for program anno-

tations. On the one hand, to synthesize invariants using fixpoint computation, program analysis often uses a finite lattice of program properties. On the other hand, program annotations should be expressed in some convenient, well-known notation, such as a variation of first-order logic. A program analysis that utilizes program specifications must bridge the gap between the analysis representation and the program annotations, for example, by providing a translation from a logic-based annotation language to the analysis representation. The translation from the full first-order logic to three-valued structures is equivalent to first-order theorem proving, and is therefore undecidable. Because we restrict our attention to formulas of a particular form, we are able to find a (complete and sound) decision procedure for generating three-valued structures that have the same meaning as these formulas.[1] The existence of this information-preserving translation algorithm indicates that our formulas have the same expressive power as three-valued structures. Nevertheless, our formulas are more flexible than the direct use of three-valued structures (or formulas isomorphic to three-valued structures). For example, our formulas may use sets that are potentially intersecting or empty, while the summary nodes of three-valued structures represent disjoint, non-empty sets of nodes.

In addition to the benefits for writing program annotations, the richer syntactic class of formulas is potentially useful for analysis representations. A set of three-valued structures corresponds to a disjunctive normal form; alternative representations for three-valued structures may be more appropriate in some cases.

2 Preliminaries

We mostly follow the setup of [46]. Let $\mathcal{A}$ be a finite set of unary relation symbols (with a typical element $A \in \mathcal{A}$) and $\mathcal{F}$ a finite set of binary relation symbols (with a typical element $f \in \mathcal{F}$). For simplicity, we consider only unary and binary relation symbols, which are usually sufficient for modelling dynamically allocated structures. A *two-valued structure* is a pair $S^\sharp = \langle U^\sharp, \iota^\sharp \rangle$ where $U^\sharp$ is a finite non-empty set (of "concrete individuals"), $\iota^\sharp(A) \in U^\sharp \to \{0,1\}$ for $A \in \mathcal{A}$, and $\iota^\sharp(f) \in (U^\sharp)^2 \to \{0,1\}$ for $f \in \mathcal{F}$. Let 2-STRUCT be the set of all two-valued structures. A *three-valued structure* is a pair $S = \langle U, \iota \rangle$ where U is a finite non-empty set (of "abstract individuals"), $\iota(A) \in U \to \{\{0\},\{1\},\{0,1\}\}$ for $A \in \mathcal{A}$ and and $\iota(f) \in U^2 \to \{\{0\},\{1\},\{0,1\}\}$ for $f \in \mathcal{F}$. Let 3-STRUCT denote the set of all three-valued structures. If $S^\sharp$ is a two-valued structure and F a closed formula in first-order logic, then $[\![F]\!]^{S^\sharp} \in \{0,1\}$ denotes the truth-value of F in $S^\sharp$, and $\gamma^*_{\mathrm{F}}(F) = \{S^\sharp \in \text{2-STRUCT} \mid [\![F]\!]^{S^\sharp} = 1\}$ is the set of models of F. If C is a set of formulas, then $\mathsf{models}[C] = \{\gamma^*_{\mathrm{F}}(F) \mid F \in C\}$ is the set of sets of models of formulas from C. Let $\mathcal{A}_1 \subseteq \mathcal{A}$ be a finite subset of unary predicates. We call elements of $\mathcal{A}_1$ *abstraction predicates*. An $\mathcal{A}_1$-bounded

[1] An alternative approach proposes the use of theorem provers to synthesize three-valued structures from arbitrary first-order formulas [49, 49, 40, 41], [48, Chapter 6].

structure is three-valued structure $\langle U, \iota \rangle$ for which the following two conditions hold: 1) $\iota(A)(u) \in \{\{0\}, \{1\}\}$ for all $A \in \mathcal{A}_1$ and all $u \in U$; 2) if $u_1, u_2 \in U$ and $u_1 \neq u_2$ then $\iota(A)(u_1) \neq \iota(A)(u_2)$ for some $A \in \mathcal{A}_1$. The following definition of tight concretization corresponds to [48, Chapter 7], [45, Definition 3.6].

Definition 1 (Tight Concretization). *Let $S^\sharp = \langle U^\sharp, \iota^\sharp \rangle$ be a two-valued structure, let $S = \langle U, \iota \rangle$ be a three-valued structure, and let $h : U^\sharp \to U$ be a surjective total function. We write $S^\sharp \sqsubseteq_T^h S$ iff*

1. *for every $A \in \mathcal{A}$ and $u \in U$: $\iota(A)(u) = \{\iota^\sharp(A)(u^\sharp) \mid h(u^\sharp) = u\}$;*
2. *for every $f \in \mathcal{F}$ and $u_1, u_2 \in U$:*

$$\iota(f)(u_1, u_2) = \{\, \iota^\sharp(f)({u_1}^\sharp, {u_2}^\sharp) \mid h({u_1}^\sharp) = u_1 \ \wedge \ h({u_2}^\sharp) = u_2\}$$

We write $S^\sharp \sqsubseteq_T S$ iff there exists a surjective total function h such that $S^\sharp \sqsubseteq_T^h S$, and in that case we call h a homomorphism. The tight concretization of a three-valued structure S, denoted $\gamma_T(S)$, is given by: $\gamma_T(S) = \{S^\sharp \mid S^\sharp \sqsubseteq_T S\}$. We extend γ_T to γ_T^* that acts on sets of three-valued structures so that the set denotes a disjunction: $\gamma_T^*(\mathcal{S}) = \bigcup_{S \in \mathcal{S}} \gamma_T(S)$. The set of sets of two-valued structures definable via three-valued structure with tight concretization is $\mathsf{models}[T_2] = \{\gamma_T^*(\mathcal{S}) \mid \mathcal{S}$ a finite set of $\mathcal{A}_1$-bounded three-valued structures$\}$. We call the set of sets $\mathsf{models}[T_2]$ boolean shape analysis constraints *(the results of this paper justify to the name).*

If $A \in \mathcal{A}$ and $\alpha \in \{0, 1\}$ then A^α is defined by $A^1 = A$ and $A^0 = \neg A$. A cube over $\mathcal{A}_1$ (or just "cube" for short) is an expression $P(x)$ of the form $A_1^{\alpha_1}(x) \wedge \ldots \wedge A_q^{\alpha_q}(x)$ where $\alpha_1, \ldots, \alpha_q \in \{0, 1\}$.

Definition 2 (TR_1-literal). *Let $P_1(x), P_2(x)$ range over cubes over $\mathcal{A}_1$, let A range over elements of $\mathcal{A} \setminus \mathcal{A}_1$, and let f range over $\mathcal{F}$. A TR_1-atomic-formula is a formula of one of the following forms:*

$\exists x.\ P_1(x)$
$\exists x.\ P_1(x) \wedge A(x)$
$\exists x.\ P_1(x) \wedge \neg A(x)$
$\exists x \exists y.\ P_1(x) \wedge P_2(y) \wedge f(x, y)$
$\exists x \exists y.\ P_1(x) \wedge P_2(y) \wedge \neg f(x, y)$

A TR_1-literal is a TR_1-atomic-formula or its negation.

TR_1-formulas correspond to formulas in [48, Chapter 7]. TR_1-formulas satisfy syntactic invariants that make them isomorphic to three-valued structures with tight-concretization semantics.

Definition 3 (TR_1-formulas). *Let $P(x)$, $P_1(x)$, $P_2(y)$ denote cubes over $\mathcal{A}_1$. A* canonical conjunction *of TR_1 literals is a conjunction of TR_1-literals that satisfies all of the following properties:*

P1. for each $P(x)$ a cube over $\mathcal{A}_1$, exactly one of the conjuncts $\exists x.P(x)$ and $\neg\exists x.P(x)$ occurs;

P2. there is at least one cube $P(x)$ such that the conjunct $\exists x.P(x)$ occurs in the conjunction;

P3. if the conjunct $\neg\exists x.P(x)$ occurs, then this conjunct is the only occurrence of the cube $P(x)$ (and the cube $P(y)$) in the conjunction;

P4. for each cube $P(x)$ such that $\exists x.P(x)$ occurs, and each $A \in \mathcal{A} \setminus \mathcal{A}_1$, exactly one of the following three conditions holds:

1. *$\neg\exists x.\ P(x) \wedge A(x)$ occurs in the conjunction,*
2. *$\neg\exists x.\ P(x) \wedge \neg A(x)$ occurs in the conjunction,*
3. *both $\exists x.\ P(x) \wedge A(x)$ and $\exists x.\ P(x) \wedge \neg A(x)$ occur in the conjunction;*

P5. for every two cubes $P_1(x)$ and $P_2(y)$ such that the conjuncts $\exists x.P_1(x)$ and $\exists x.P_2(x)$ occur, and for every $f \in \mathcal{F}$, exactly one one of the following three conditions holds:

1. *$\neg\exists x\exists y.\ P_1(x) \wedge P_2(y) \wedge f(x,y)$ occurs in the conjunction;*
2. *$\neg\exists x\exists y.\ P_1(x) \wedge P_2(y) \wedge \neg f(x,y)$ occurs in the conjunction;*
3. *both $\exists x\exists y.\ P_1(x) \wedge P_2(y) \wedge f(x,y)$ and $\exists x\exists y.\ P_1(x) \wedge P_2(y) \wedge \neg f(x,y)$ occur in the conjunction.*

A TR_1-formula is a disjunction of canonical conjunctions of TR_1-literals.

A small difference between TR_1-formulas and formulas in [48, Definition 7.3.3, Page 31] is that [48, Definition 7.3.3, Page 31] does not contain conjuncts of the form $\neg\exists x.P(x)$ stating the emptiness of each empty cube, but instead contains one conjunct of the form $\forall x. \bigvee_P P(x)$ where P ranges over all non-empty cubes.

The following Proposition 4 shows that TR_1 formulas capture precisely the meaning of three-valued structures under tight concretization. The proof of Proposition 4 was first presented in [48, Appendix B] (and reviewed in [31, Page 9]). The proof shows that TR_1 formulas and bounded three-valued structures can be viewed as different notations for the same mathematical structure.

Proposition 4. $\mathsf{models}[TR_1] = \mathsf{models}[T_2]$

3 A New Characterization of Three-Valued Structures

Definition 5 introduces the new syntactic class of formulas characterizing three-valued structures under tight concretization semantics. Theorem 6 gives a constructive proof of the correctness of the characterization.

Definition 5 (TR_4-formulas). *Let $B_1(x), B_2(y)$ be range over arbitrary boolean combinations of elements of $\mathcal{A}_1$, let $Q(x)$ range over disjunctions of literals of form $A(x)$ and $\neg A(x)$ where $A \in \mathcal{A} \setminus \mathcal{A}_1$, and let $g(x,y)$ range over disjunctions of literals of the form $f(x,y)$ and $\neg f(x,y)$ where $f \in \mathcal{F}$.*

A TR_4-atomic-formula is a formula of one of the following forms:

1. $\exists x.\ B_1(x)$
2. $\exists x.\ B_1(x) \wedge Q(x)$
3. $\exists x\exists y.\ B_1(x) \wedge B_2(y) \wedge g(x,y)$

$$\exists x.B_1(x) \vee B_2(x) \;\rightarrow\; (\exists x.B_1(x)) \vee (\exists x.B_2(x))$$

$$\exists x.\ (B_1(x) \vee B_2(x)) \wedge Q(x) \;\rightarrow\; (\exists x.B_1(x) \wedge Q(x)) \vee (\exists x.B_2(x) \wedge Q(x))$$

$$\exists x.\ B_1(x) \wedge (Q_1(x) \vee Q_2(x)) \;\rightarrow\; (\exists x.B_1(x) \wedge Q_1(x)) \vee (\exists x.B_1(x) \wedge Q_2(x))$$

$$\exists x \exists y.\ (B_{11}(x) \vee B_{12}(x)) \wedge B_2(y) \wedge g(x,y) \rightarrow \exists x \exists y.\ B_{11}(x) \wedge B_2(y) \wedge g(x,y) \vee \exists x \exists y.\ B_{12}(x) \wedge B_2(y) \wedge g(x,y)$$

$$\exists x \exists y.\ B_1(x) \wedge (B_{21}(y) \vee B_{22}(y)) \wedge g(x,y) \rightarrow \exists x \exists y.\ B_1(x) \wedge B_{21}(y) \wedge g(x,y) \vee \exists x \exists y.\ B_1(x) \wedge B_{22}(y) \wedge g(x,y)$$

$$\exists x \exists y.\ B_1(x) \wedge B_2(y) \wedge (g_1(x,y) \vee g_2(x,y)) \rightarrow \exists x \exists y.\ B_1(x) \wedge B_2(y) \wedge g_1(x,y) \wedge \exists x \exists y.\ B_1(x) \wedge B_2(y) \wedge g_2(x,y)$$

Fig. 1. Transforming TR_4-literals into TR_1-literals.

Ensure each of the Properties of Definition 3 by applying the appropriate rules:

P1. $(\exists x.P(x)) \wedge (\neg \exists x.P(x)) \rightarrow \mathsf{false}$
$\mathsf{true} \rightarrow (\exists x.P(x)) \vee (\neg \exists x.P(x))$

P2. $\bigwedge_{P \in \mathsf{cubes}} \neg \exists x.P(x) \rightarrow \mathsf{false}$

P3. $(\neg \exists x.P(x)) \wedge (\exists x.P(x) \wedge Q(x)) \rightarrow \mathsf{false}$
$(\neg \exists x.P(x)) \wedge (\exists x \exists y.P(x) \wedge Q(x,y)) \rightarrow \mathsf{false}$
$(\neg \exists x.P(x)) \wedge (\exists x \exists y.P(y) \wedge Q(x,y)) \rightarrow \mathsf{false}$
$(\neg \exists x.P(x)) \wedge (\neg \exists x.P(x) \wedge Q(x)) \rightarrow \neg \exists x.P(x)$
$(\neg \exists x.P(x)) \wedge (\neg \exists x \exists y.P(x) \wedge Q(x,y)) \rightarrow \neg \exists x.P(x)$
$(\neg \exists x.P(x)) \wedge (\neg \exists x \exists y.P(y) \wedge Q(x,y)) \rightarrow \neg \exists x.P(x)$

P4. $(\exists x.P(x) \wedge Q(x)) \wedge (\neg \exists x.P(x) \wedge Q(x)) \rightarrow \mathsf{false}$
$(\neg \exists x.P(x) \wedge A(x)) \wedge (\neg \exists x.P(x) \wedge \neg A(x)) \;\rightarrow\; \neg \exists x.P(x)$
$\mathsf{true} \rightarrow (\neg \exists x.P(x) \wedge A(x)) \vee (\neg \exists x.P(x) \wedge \neg A(x)) \vee (\exists x.P(x) \wedge A(x)) \wedge (\exists x.P(x) \wedge \neg A(x))$
+rules for $P3$

P5. $(\exists x \exists y.P_1(x) \wedge P_2(y) \wedge Q(x,y)) \wedge (\neg \exists x \exists y.P_1(x) \wedge P_2(y) \wedge Q(x,y)) \rightarrow \mathsf{false}$
$(\neg \exists x \exists y.P_1(x) \wedge P_2(y) \wedge f(x,y)) \;\wedge\; (\neg \exists x \exists y.P_1(x) \wedge P_2(y) \wedge \neg f(x,y)) \;\rightarrow\; (\neg \exists x.P_1(x)) \vee (\neg \exists y.P_2(y))$
$\mathsf{true} \rightarrow (\neg \exists x \exists y.P_1(x) \wedge P_2(y) \wedge f(x,y)) \vee (\neg \exists x \exists y.P_1(x) \wedge P_2(y) \wedge \neg f(x,y)) \vee (\exists x \exists y.P_1(x) \wedge P_2(y) \wedge f(x,y)) \;\wedge\; (\exists x \exists y.P_1(x) \wedge P_2(y) \wedge \neg f(x,y))$
+rules for $P3$

Fig. 2. Transforming a conjunction of TR_1-literals into a canonical conjunction of TR_1-literals.

1. apply the rules in Figure 1 to transform the TR_4-formula into a boolean combination of TR_1-literals;
2. transform the formula into a disjunction of conjunctions of TR_1-literals;
3. apply the rules in Figure 2 to transform each conjunction of TR_1-literals into a canonical conjunction of TR_1-literals, while keeping the formula in disjunctive normal form.

Fig. 3. Normalization algorithm for transforming TR_4-formulas into TR_1-formulas.

A TR_4-literal is a TR_4-atomic-formula or its negation. A TR_4-formula is a boolean combination of TR_4-atomic-formulas.

Theorem 6. *Algorithm sketched in Figures 3, 1, 2 converts a TR_4-formula into an equivalent TR_1-formula in a finite number of steps.*

Corollary 7. $\mathsf{models}[TR_4] = \mathsf{models}[TR_1] = \mathsf{models}[T_2]$.

By definition, TR_4-formulas are closed under all boolean operations.

Corollary 8. *1. The family of sets* $\mathsf{models}[T_2]$ *forms a boolean algebra of sets which is a subalgebra of the boolean algebra of all subsets of* 2-STRUCT.
2. There is an algorithm that constructs, given two finite sets of bounded three-valued structures $\mathcal{S}_1$ and $\mathcal{S}_2$, a finite set of bounded three-valued structures $\mathcal{S}_3$ such that $\gamma^*_T(\mathcal{S}_1) \subseteq \gamma^*_T(\mathcal{S}_2)$ iff $\gamma^*_T(\mathcal{S}_3) = \emptyset$.
3. There is an algorithm that constructs, given two finite sets of bounded three-valued structures $\mathcal{S}_1$ and $\mathcal{S}_2$, a finite set of bounded three-valued structures $\mathcal{S}_3$ such that: $\gamma^*_T(\mathcal{S}_1) = \gamma^*_T(\mathcal{S}_2)$ iff $\gamma^*_T(\mathcal{S}_3) = \emptyset$.

Note. Every TR_4-formula with the set of abstraction predicates $\mathcal{A}_1 \subseteq \mathcal{A}$ is also a TR_4 formula with the set of abstraction predicates $\mathcal{A}_1 = \mathcal{A}$. When $\mathcal{A} = \mathcal{A}_1$, then the class of TR_4-formulas can be defined simply as boolean combinations of formulas 1) $\exists x.\ B_1(x)$, and 2) $\exists x \exists y.\ B_1(x) \wedge B_2(y) \wedge g(x, y)$ where $B_1(x)$, $B_2(y)$ are boolean combinations of literals of the form $A(x)$ and $A(y)$ for $A \in \mathcal{A}$, and $g(x, y)$ ranges over disjunctions of literals of the form $f(x, y)$ and $\neg f(x, y)$ for $f \in \mathcal{F}$.

4 Decidability of Independent Predicates

We next examine the decidability of the questions of the form: "Given sets $\mathcal{A}$ and $\mathcal{F}$ and a TR_1-formula F over predicates $\mathcal{A}$ and $\mathcal{F}$, is F satisfiable?" (Note that the sets $\mathcal{A}$ and $\mathcal{F}$ are part of the input to the decision procedure; for fixed finite sets $\mathcal{A}$ and $\mathcal{F}$ there are finitely many three-valued structures, so the decision problem would be trivial.) It turns out that satisfiability of TR_1-formulas over 2-STRUCT is decidable because the family of TR_1-formulas over 2-STRUCT has small model property. It is easy to construct a model $\langle U^\sharp, \iota^\sharp \rangle$ of a canonical conjunction of TR_1-literals by introducing at most two elements of the domain $U^\sharp$ for each non-empty cube.

Proposition 9. *Let F be a canonical conjunction of TR_1-literals and let the number of cubes $P(x)$ over $\mathcal{A}_1$ such that $\exists x.P(x)$ occurs in F be n. Then there exists a two-valued structure $S^\sharp = \langle U^\sharp, \iota^\sharp \rangle$ such that $|U^\sharp| = 2n$ and F is true in $S^\sharp$.*

Corollary 10. *$\gamma_T^*(\mathcal{S}) = \emptyset$ iff $\mathcal{S} = \emptyset$.*

Corollary 11. *The following questions are decidable for sets $\mathcal{S}_1, \mathcal{S}_2$ of three-valued structures: 1) $\gamma_T^*(\mathcal{S}_1) = \emptyset$; 2) $\gamma_T^*(\mathcal{S}_1) \subseteq \gamma_T^*(\mathcal{S}_2)$; 3) $\gamma_T^*(\mathcal{S}_1) = \gamma_T^*(\mathcal{S}_2)$.*

5 Structures with Defined Predicates

Previous sections interpret three-valued structures and formulas over the set 2-STRUCT of *all* two-valued structures. In general, it is useful to interpret three-valued structures and formulas over some subset 2-CSTRUCT $\subseteq$ 2-STRUCT of *compatible* two-valued structures [46, Page 268]. The meaning of tight concretization with respect to 2-CSTRUCT is $c\gamma_T^*(\mathcal{S}) = \gamma_T^*(\mathcal{S}) \cap$ 2-CSTRUCT and we let $\mathsf{models}[cT_2]$ denote the set of all $c\gamma_T^*(\mathcal{S})$ for all finite sets of bounded three-valued structures. To characterize the meaning of three-valued structures over 2-CSTRUCT, for each class of formulas TR_i we introduce the corresponding class cTR_i by conjoining the formulas with a first-order formula F_ψ that characterizes the subset 2-CSTRUCT. It then follows $\mathsf{models}[cTR_i] = \{\mathcal{S}^\sharp \cap \text{2-CSTRUCT} \mid \mathcal{S}^\sharp \in \mathsf{models}[TR_i]\}$. Hence, $\mathsf{models}[cTR_4]$ is a subalgebra of the boolean algebra of subsets of 2-CSTRUCT, its sets are subsets of elements of $\mathsf{models}[TR_4]$, and the following generalization of Corollary 11 holds.

Corollary 12. *Assume that the satisfiability for three-valued structures interpreted over* 2-CSTRUCT *using tight concretization is decidable. Then the entailment and the equivalence of three-valued structures interpreted over* 2-CSTRUCT *using tight concretization are also decidable.*

6 Further Related Work

Researchers have proposed several program checking techniques based on dataflow analysis and symbolic execution [13, 19, 14, 8, 10, 6, 35]. The primary strength of shape analysis compared to the alternative approaches is the ability to perform sound and precise reasoning about dynamically allocated data structures.

Our work follows the line of shape analysis approaches which view the program as operating on concrete graph structures [46,22,9,32,20,16,15,37,27]. An alternative approach is to identify each heap object using the set of paths that lead to the object [12,18,7]. Other notations for reasoning about the heap include spatial logic [21] and alias types [47]. In the past we have seen a contrast between the approach to verification of dynamically allocated data structures based on

Hoare logic [37, 21, 14], and the approach based on manipulation of graph summaries [32, 9, 22, 16]. The work [20] and especially [45] are important steps in bringing these two views together. Along with the recent work [40, 49, 50, 48] our paper makes further contributions to unifying these two approaches.

The parametric framework for shape analysis is presented in [45, 46]. A systematic presentation of three-valued logic with equality is given in [38]. A description of a three-valued logic analyzer TVLA is in [33], an extension to interprocedural analysis is in [43, 42], and the use of shape analysis for program verification is demonstrated in [34]. A finite differencing approach for automatically computing transfer functions for analysis is presented in [39]. A shape analysis tool must ultimately take into account the definitions of instrumentation predicates, which requires some form of theorem proving or decision procedures. The original work [46, Page 272] uses rules based on Horn clauses for such reasoning, whereas [40, 49, 50, 48] (see Section 1) propose the use of theorem provers and decision procedures. In this paper we have identified one component of the problem that is always decidable and useful: it is always possible to reduce entailment and equivalence problems to the satisfiability problem. Of great importance for taking advantage of our result, as well as the results of [40, 49, 50, 48], are decidable logics that can express heap properties. Among the promising such logics are monadic second-order logic of trees [23], the logic L_r [4], and role logic [30].

It is possible to apply predicate abstraction techniques [3, 2, 17] to perform shape analysis; the view of three-valued structures as boolean combinations of constraints of certain form may be beneficial for this direction of work and enable the easier application of representations such as binary decision diagrams [11, 5, 36]. The boolean algebra of state predicates and predicate transformers has been used successfully as the foundation of refinement calculus [1]. In this paper we have identified a particular subalgebra of the boolean algebra of all state predicates; we view this boolean algebra as providing the foundation of shape analysis.

7 Conclusions and Future Work

We have presented a new characterization of the constraints used as dataflow facts in parametric shape analysis. Our characterization represents these dataflow facts as boolean combinations of formulas. Among the useful consequences of the closure of boolean shape analysis constraints under all boolean operations is the fact that the entailment and the equivalence of these constraints is reducible to the satisfiability of the constraints.

In this paper we have focused on the tight-concretization semantics of three-valued structures. In the full version of the paper [31] we additionally show similar results for standard-concretization semantics of three-valued structures, with one important difference: the resulting constraints are closed under conjunction and disjunction, but not under negation. In fact, the least boolean algebra generated by those constraints is precisely the boolean algebra of boolean shape analysis constraints.

We view the results of this paper as a step in further understanding of the foundations of shape analysis. To make the connection with [46], this paper starts with three-valued structures and proceeds to characterize the structures using formulas. An alternative approach is to start with formulas that express the desired properties and then explore efficient ways of representing and manipulating these formulas. We believe that the entire framework [46] can be reformulated using canonical forms of formulas instead of three-valued structures. We also expect that the idea of viewing dataflow facts as canonical forms of formulas is methodologically useful in general, especially for the analyses that verify complex program properties.

Acknowledgements. The results of this paper were inspired in part by the discussions with Andreas Podelski, Thomas Reps, Mooly Sagiv, Greta Yorsh, and David Schmidt. We thank Greta Yorsh and Darko Marinov for useful comments on an earlier version of the paper.

References

1. Ralph-Johan Back and Joakim von Wright. *Refinement Calculus*. Springer-Verlag, 1998.
2. Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. ACM PLDI*, 2001.
3. Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Relative completeness of abstraction refinement for software model checking. In *TACAS'02*, volume 2280 of *LNCS*, page 158, 2002.
4. Michael Benedikt, Thomas Reps, and Mooly Sagiv. A decidable logic for linked data structures. In *Proc. 8th ESOP*, 1999.
5. Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using BDDs. In *PLDI 2003*, 2003.
6. Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A Static Analyzer for Large Safety-Critical Software. In *ACM PLDI*, San Diego, California, June 2003. ACM.
7. Marius Bozga, Radu Iosif, and Yassine Laknech. Storeless semantics and alias logic. In *ACM PEPM'03*, pages 55–65. ACM Press, 2003.
8. William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software—Practice & Experience*, 30(7):775–802, 2000.
9. David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proc. ACM PLDI*, 1990.
10. Lori Clarke and Debra Richardson. Symbolic evaluation methods for program analysis. In *Program Flow Analysis: Theory and Applications*, chapter 9. Prentice-Hall, Inc., 1981.
11. Dennis Dams and Kedar S. Namjoshi. Shape analysis through predicate abstraction and model checking. In *VMCAI 2003*, volume 2575 of *LNCS*, pages 310–323, 2003.
12. Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 230–241. ACM Press, 1994.

13. Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proc. 4th USENIX OSDI*, 2000.
14. Cormac Flanagan, K. Rustan M. Leino, Mark Lilibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In *Proc. ACM PLDI*, 2002.
15. Pascal Fradet and Daniel Le Métayer. Shape types. In *Proc. 24th ACM POPL*, 1997.
16. Rakesh Ghiya and Laurie Hendren. Is it a tree, a DAG, or a cyclic graph? In *Proc. 23rd ACM POPL*, 1996.
17. Susanne Graf and Hassen Saidi. Construction of abstract state graphs with pvs. In *Proc. 9th CAV*, pages 72–83, 1997.
18. C. A. R. Hoare and He Jifeng. A trace model for pointers and objects. In *Proc. 13th ECOOP*, volume 1628 of *LNCS*, 1999.
19. Gerard J. Holzmann. Static source code checking for user-defined properties. In *Proc. IDPT 2002, Pasadena, CA*, June 2002.
20. Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *Proc. ACM PLDI*, 1994.
21. Samin Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Proc. 28th ACM POPL*, 2001.
22. N. D. Jones and S. S. Muchnick. Flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Proc. 9th ACM POPL*, 1982.
23. Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. MONA implementation secrets. In *Proc. 5th International Conference on Implementation and Application of Automata*. LNCS, 2000.
24. Stephen Cole Kleene. *Introduction to Metamathematics*. D. Van Nostrand Company, Inc., Princeton, New Jersey, 1952. fifth reprint, 1967.
25. Viktor Kuncak, Patrick Lam, and Martin Rinard. A language for role specifications. In *Proceedings of the 14th Workshop on Languages and Compilers for Parallel Computing*, volume 2624 of *Lecture Notes in Computer Science, Springer*, 2001.
26. Viktor Kuncak, Patrick Lam, and Martin Rinard. Roles are really great! Technical Report 822, Laboratory for Computer Science, Massachusetts Institute of Technology, 2001.
27. Viktor Kuncak, Patrick Lam, and Martin Rinard. Role analysis. In *Proc. 29th POPL*, 2002.
28. Viktor Kuncak and Martin Rinard. Typestate checking and regular graph constraints. Technical Report 863, MIT Laboratory for Computer Science, 2002.
29. Viktor Kuncak and Martin Rinard. Existential heap abstraction entailment is undecidable. In *10th Annual International Static Analysis Symposium (SAS 2003)*, San Diego, California, June 11–13 2003.
30. Viktor Kuncak and Martin Rinard. On role logic. Technical Report 925, MIT CSAIL, 2003.
31. Viktor Kuncak and Martin Rinard. On the boolean algebra of shape analysis constraints. Technical report, MIT CSAIL, August 2003.
32. James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. In *Proc. ACM PLDI*, Atlanta, GA, June 1988.
33. Tal Lev-Ami. TVLA: A framework for kleene based logic static analyses. Master's thesis, Tel-Aviv University, Israel, 2000.

34. Tal Lev-Ami, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Putting static analysis to work for verification: A case study. In *International Symposium on Software Testing and Analysis*, 2000.
35. Francesco Logozzo. Class-level modular analysis for object oriented languages. In *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11–13, 2003, Proceedings*, volume 2694 of *Lecture Notes in Computer Science*. Springer, 2003.
36. Roman Manevich, G. Ramalingam, John Field, Deepak Goyal, and Mooly Sagiv. Compactly representing first-order structures for static analysis. In *Proc. 9th International Static Analysis Symposium*, pages 196–212, 2002.
37. Anders Møller and Michael I. Schwartzbach. The Pointer Assertion Logic Engine. In *Proc. ACM PLDI*, 2001.
38. Flemming Nielson, Hanne Riis Nielson, and Mooly Sagiv. Kleene's logic with equality. *Information Processing Letters*, 80(3):131–137, 2001.
39. Thomas Reps, Mooly Sagiv, and Alexey Loginov. Finite differencing of logical formulas for static analysis. In *Proc. 12th ESOP*, 2003.
40. Thomas Reps, Mooly Sagiv, and Greta Yorsh. Symbolic implementation of the best transformer. Technical Report TR-1468, University of Wisconsin, January 2003.
41. Thomas Reps, Mooly Sagiv, and Greta Yorsh. Symbolic implementation of the best transformer. In *VMCAI'04*, 2004.
42. Noam Rinetzky. Interprocedural shape analysis. Master's thesis, Technion - Israel Institute of Technology, 2000.
43. Noam Rinetzky and Mooly Sagiv. Interprocedual shape analysis for recursive programs. In *Proc. 10th International Conference on Compiler Construction*, 2001.
44. Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM TOPLAS*, 20(1):1–50, 1998.
45. Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *Proc. 26th ACM POPL*, 1999.
46. Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3):217–298, 2002.
47. Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *Proc. 9th ESOP*, Berlin, Germany, March 2000.
48. Greta Yorsh. Logical characterizations of heap abstractions. Master's thesis, Tel-Aviv University, March 2003.
49. Greta Yorsh, Thomas Reps, and Mooly Sagiv. Symbolically computing most-precise abstract operations for shape analysis. Technical report, Tel-Aviv University, September 2003. Forthcoming.
50. Greta Yorsh, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Logical characterizations of heap abstractions. Technical report, University of Wisconsin, Madison, January 2003.

Appendix: Example

Figure 4 illustrates the use of boolean shape analysis constraints and their closure properties. The left column introduces a set of constraints that provide a partial specification of an operation that removes some elements from a list. The right column shows that the conjunction of these constraints is a TR_4-formula. In this example $\mathcal{A}_1 = \mathcal{A} = \{A_l, A_r, A_{r0}\}$ and $\mathcal{F} = \{f\}$. The predicate A_l represents the

object referenced by a local variable. The predicate A_{r0} denotes the set of nodes reachable from the local variable in the initial state, whereas the predicate A_r denotes the set of nodes reachable from the local variable in the final state of the operation. The binary relation f represents the value of the "next" pointer of the list in the final state. The meaning of the constraints is the following: C_1) the first element of the list has no incoming references; C_2) the list has at least two elements; C_3) the object referenced by local variable is reachable from the local variable in both pre- and post- state; C_4) following reachable nodes along the f field yields reachable nodes; C_5) all nodes are reachable; C_6) the data structure operation only removes elements from the set, it does not add any elements.

Consider the question whether the formula C_7 in Figure 4 is a consequence of the conjunction of constraints $\bigwedge_{i=1}^{6} C_i$. Transform the formula $\neg C_7 \wedge \bigwedge_{i=1}^{6} C_i$ into a TR_1-formula using our normalization algorithm in Figure 3. The result is the set of counterexamples in Figure 5 (represented as three-valued structures). These counterexamples show that the formula C_7 is *not* a consequence of the constraints in Figure 4, and show the set of scenarios in which the violation of formula C_7 can occur.

$$
\begin{array}{ll}
C_1 : \forall y.A_l(y) \Rightarrow \neg\exists x.f(x,y) & \neg\exists x.\exists y.A_l(y) \wedge f(x,y) \\
C_2 : \exists x\exists y.A_l(x) \wedge \neg A_l(y) \wedge f(x,y) & \exists x\exists y.A_l(x) \wedge \neg A_l(y) \wedge f(x,y) \\
C_3 : \forall x.A_l(x) \Rightarrow A_r(x) \wedge A_{r0}(x) & \neg\exists x.\neg(A_l(x) \Rightarrow A_r(x) \wedge A_{r0}(x)) \\
C_4 : \forall x\forall y.A_r(x) \wedge f(x,y) \Rightarrow A_r(y) & \neg\exists x\exists y.A_r(x) \wedge \neg A_r(y) \wedge f(x,y) \\
C_5 : \forall x.A_l(x) \vee A_r(x) \vee A_{r0}(x) & \neg\exists x.\neg(A_l(x) \vee A_r(x) \vee A_{r0}(x)) \\
C_6 : \forall x.A_r(x) \Rightarrow A_{r0}(x) & \neg\exists x.\neg(A_r(x) \Rightarrow A_{r0}(x))
\end{array}
$$

$$
C_7 : \qquad \forall x\forall y.\neg A_l(x) \wedge \neg A_r(x) \wedge f(x,y) \Rightarrow \neg A_r(y)
$$

Fig. 4. Example verification of entailment of boolean shape analysis constraints.

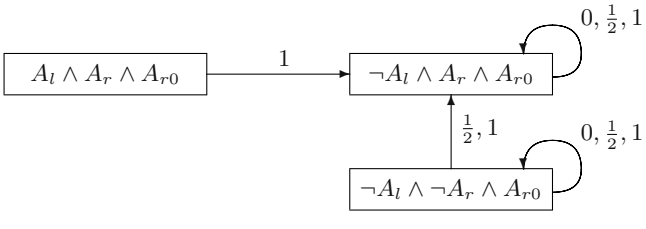


Fig. 5. Counterexample structures for the entailment of constraints in Figure 4. There are $2 \cdot 3 \cdot 3$ counterexample three-valued structures, for different values of edges.

Approximate Probabilistic Model Checking

Thomas Hérault[1], Richard Lassaigne[2], Frédéric Magniette[1], and Sylvain Peyronnet[1]

[1] LRI, University Paris XI
{herault,magniett,syp}@lri.fr
[2] University Paris VII
lassaign@logique.jussieu.fr

Abstract. Symbolic model checking methods have been extended recently to the verification of probabilistic systems. However, the representation of the transition matrix may be expensive for very large systems and may induce a prohibitive cost for the model checking algorithm. In this paper, we propose an approximation method to verify quantitative properties on discrete Markov chains. We give a randomized algorithm to approximate the probability that a property expressed by some positive LTL formula is satisfied with high confidence by a probabilistic system. Our randomized algorithm requires only a succinct representation of the system and is based on an execution sampling method. We also present an implementation and a few classical examples to demonstrate the effectiveness of our approach.

1 Introduction

In this paper, we address the problem of verifying quantitative properties on discrete time Markov chains (DTMC). We present an efficient procedure to approximate model checking of positive LTL formulas on probabilistic transition systems. This procedure decides if the probability of a formula over the whole system is greater than a certain threshold by sampling finite execution paths. It allows us to verify monotone properties on the system with high confidence. For example, we can verify a property such as : "the probability that the message sent will be received without error is greater than 0.99". This method is an improvement on the method described in [16].

The main advantage of this approach is to allow verification of formulas even if the transition system is huge, even without any abstraction. Indeed, we do not have to deal with the state space explosion phenomenon because we verify the property on only one finite execution path at a time. This approach can be used in addition to classical probabilistic model checkers when the verification is intractable.

Our main results are:

– A method that allows the efficient approximation of the satisfaction probability of monotone properties on probabilistic systems.

B. Steffen and G. Levi (Eds.): VMCAI 2004, LNCS 2937, pp. 73–84, 2004.

– A tool named APMC that implements the method. We use it to verify extremely large systems such as the Pnueli and Zuck's 500 dining philosophers.

The paper is organized as follows. In Section 2, we review related work on probabilistic verification of qualitative and quantitative properties. In Section 3, we consider fully probabilistic systems and classical LTL logic. In Section 4, we explain how to adapt the main idea of the bounded model checking approach to the probabilistic framework. In Section 5, we present a randomized algorithm for the approximation of the satisfaction probability of monotone properties. In Section 6, we present our tool and give experimental results and compare them with the probabilistic model checker PRISM [7].

2 Related Work

Several methods have been proposed to verify a probabilistic or a concurrent probabilistic system against LTL formulas. Vardi and Wolper [26,27] developed an automata theoretical approach for verifying qualitative properties stating that a linear time formula holds with probability 0 or 1. Pnueli and Zuck [22] introduced a model checking method for this problem.

Courcoubetis and Yannakakis [5] studied probabilistic verification of quantitative properties expressed in the linear time framework. For the fully probabilistic case, the time complexity of their method is polynomial in the size of the state space, and exponential in the size of the formula. For the concurrent case, the time complexity is linear in the size of the system, and double exponential in the size of the formula.

Hansson and Jonsson [9] introduced the logic $PCTL$ (Probabilistic Computation Tree Logic) and proposed a model checking algorithm for fully probabilistic systems. They combined reachability-based computation, as in classical model checking, and resolution of systems of linear equations to compute the probability associated with the until operator. For concurrent probabilistic systems, Bianco and de Alfaro [3] showed that the minimal and maximal probabilities for the until operator can be computed by solving linear optimization problems. The time complexity of these algorithms are polynomial in the size of the system and linear in the size of the formula.

There are a few model checking tools that are designed for the verification of quantitative specifications. ProbVerus [8] uses $PCTL$ model checking and symbolic techniques to verify $PCTL$ formulas on fully probabilistic systems. PRISM [7,15] is a probabilistic symbolic model checker that can check $PCTL$ formulas on fully or concurrent probabilistic systems. Reachability-based computation is implemented using BDDs, and numerical analysis may be performed by a choice between three methods: MTBDD-based representation of matrices, conventional sparse matrices, or a hybrid approach. The *Erlangen-Twente Markov Chain Checker* [10] ($E \vdash MC^2$) supports model checking of continuous-time Markov chains against specifications expressed in continuous-time stochastic logic (CSL). Rapture, presented in [6] and [12] uses abstraction and refinement to check a subset of $PCTL$ over concurrent probabilistic systems.

In [28], Younes and Simmons described a procedure for verifying properties of discrete event systems based on Monte-Carlo simulation and statistical hypothesis testing. This procedure uses a refinement technique to build statistical tests for the satisfaction probability of CSL formulas. Their logic framework is more general than ours, but they cannot predict the sampling size, in contrast with our approximation method in which this size is exactly known and tractable. Rabinovich [24] gives an algorithm to calculate the probability that a property of a probabilistic lossy channel system is satisfied. Monniaux [18] defined abstract interpretation of probabilistic programs to obtain over-approximations for probability measures. We use a similar Monte-Carlo method to approximate quantitative properties.

3 Probabilistic Transition Systems

In this section, we introduce the classical concepts for the verification of probabilistic systems.

Definition 1. *A Discrete Time Markov Chain (DTMC) is a pair* $\mathcal{M} = (S, P)$ *where* S *is a finite or enumerable set of states and* $P : S \times S \to [0, 1]$ *is a transition probability function, i.e. for all* $s \in S$, $\sum_{t \in S} P(s, t) = 1$. *If* S *is finite, we can consider* P *to be a transition matrix.*

The notion of DTMC can be extended to the notion of probabilistic transition system by adding a labeling function.

Definition 2. *A fully probabilistic transition system (PTS) is a structure* $\mathcal{M} = (S, P, I, L)$ *where (S,P) is a DTMC, I is the set of initial states and* $L : S \to \mathcal{P}(AP)$ *a function which labels each state with a set of atomic propositions.*

Definition 3. *A path* σ *of a PTS is a finite or infinite sequence of states* $(s_0, s_1, \ldots, s_i, \ldots)$ *such that* $P(s_i, s_{i+1}) > 0$ *for all* $i \geq 0$.

We denote by $Path(s)$ the set of paths whose first state is s. We note also $\sigma(i)$ the $(i+1)$-th state of path σ and σ^i the path $(\sigma(i), \sigma(i+1), \ldots)$. The length of a path σ is the number of states in the path and is denoted by $|\sigma|$, this length can be infinite.

Definition 4. *For each PTS* $\mathcal{M}$ *and state* s, *we may define a probability measure Prob on the set* $Path(s)$. *Prob denotes here the unique probability measure on the Borel field of sets generated by the basic cylinders*
$\{\sigma / \sigma$ *is a path and*$(s_0, s_1, \ldots, s_n)$ *is a prefix of* $\sigma\}$ *where*
$Prob(\{\sigma / \sigma$ *is a path and*$(s_0, s_1, \ldots, s_n)$ *is a prefix of* $\sigma\}) = \prod_{i=1}^{n} P(s_{i-1}, s_i)$.

Definition 5. *Let* σ *be a path of length* k *in a PTS* $\mathcal{M}$. *The satisfaction of a LTL formula on* σ *is defined as follows:*

- $\mathcal{M}, \sigma \models a$ *iff* $a \in L(\sigma(0))$.
- $\mathcal{M}, \sigma \models \neg\phi$ *iff* $\mathcal{M}, \sigma \not\models \phi$.
- $\mathcal{M}, \sigma \models \phi \wedge \psi$ *iff* $\mathcal{M}, \sigma \models \phi$ *and* $\mathcal{M}, \sigma \models \psi$.

- $\mathcal{M}, \sigma \models \mathbf{X}\phi$ *iff* $\mathcal{M}, \sigma^1 \models \phi$ *and* $\mid \sigma \mid > 0$.
- $\mathcal{M}, \sigma \models \phi \mathbf{U} \psi$ *iff there exists* $0 \leq j \leq k$ *s.t.* $\mathcal{M}, \sigma^j \models \psi$ *and for all* $i < j$ $\mathcal{M}, \sigma^i \models \phi$.

We now introduce a fragment of LTL which expresses only monotone properties.

Definition 6. *The essentially positive fragment (EPF) of LTL is the set of formulas built from atomic formulas (p), their negations (¬p), closed under* $\vee$, $\wedge$ *and the temporal operators* X, U.

Definition 7. *Let* $Path_k(s)$ *be the set of all paths of length k in a PTS starting at* $s \in I$. *The probability of a LTL formula* ϕ *on* $Path_k(s)$ *is the measure of paths satifying* ϕ *(as stated in Definition 5) in* $Path_k(s)$.

Definition 8. *An LTL formula* ϕ *is said to be* monotone *if and only if for all k, for all paths* σ *of length k,* $\mathcal{M}, \sigma \models \phi \implies \mathcal{M}, \sigma^+ \models \phi$, *where* σ^+ *is any path of which* σ *is a prefix.*

In [26], it is shown that for any LTL formula ϕ, probabilistic transition system $\mathcal{M}$ and state s, the set of paths $\{\sigma / \sigma(0) = s$ and $\mathcal{M}, \sigma \models \phi\}$ is measurable. We denote by $Prob[\phi]$ the measure of this set.

4 Probabilistic Bounded Model Checking

In this section, we review the classical framework for bounded model checking of linear time temporal formulas over transition systems. Then, we show that we cannot directly extend this approach but we use the main idea of checking formulas on paths of bounded length to approximate the target satisfaction probability.

Biere, Cimatti, Clarke and Zhu [4] present a symbolic model checking technique based on SAT procedures instead of BDDs. They introduce bounded model checking (BMC), where the bound correspond to the maximal length of a possible counterexample. First, they give a correspondence between BMC and classical model checking. Then they show how to reduce BMC to propositional satisfiability in polynomial time.

To check the initial property ϕ, one should look for the existence of a counterexample to ϕ, that is a path satisfying $\psi = \neg\phi$ for a given length k. In [4], the following result is also stated: if one does not find such a counterexample for $k \leq |S| \times 2^{|\psi|}$, where S is the set of states, then the initial property is true. We cannot hope to find a polynomial bound on k with respect to the size of S and ψ unless NP=PSPACE, since the model checking problem for LTL is PSPACE-complete (see [25]) and such a bound would yield a polynomial reduction to propositional satisfiability.

We try to check $Prob[\psi] \geq b$ by considering $Prob_k[\psi] \geq b$, i.e., on the probabilistic space limited to the set of paths of length k. Following the BMC approach, we could associate to a formula ψ and length k a propositional formula ψ_k in

such a way that a path of length k satisfying ψ corresponds to an assignment satisfying ψ_k. Thus determining $Prob_k[\psi]$ could be reduced to the problem of counting the number of assignments satisfying a propositional formula, called # SAT [20]. Unfortunately, not only are no efficient algorithms known for such counting problems, but they are believed to be strongly intractable (see, for instance [20]). However, it is not necessary to do such a transformation since we can evaluate directly the formula on one finite path. In the following, we use this straightforward evaluation instead of SAT-solving methods.

For many natural formulas, truth at length k implies truth in the entire model. These formulas are the so-called monotone formulas (see definition 8). We consider the subset (EPF definition 6) of LTL formulas which have this property.

EPF formulas include nested compositions of U but do not allow for negations in front. Nevertheless, this fragment can express various classical properties of transition systems such as reachability, livelock-freeness properties and convergence properties of protocols.

Proposition 1. *Let ϕ be a LTL formula. If $\phi \in EPF$, then ϕ is monotone.*

The proof of this proposition is immediate from the structure of the formula.

The monotonicity of the property defined by an EPF formula gives the following result.

Proposition 2. *For any formula ϕ of the essentially positive fragment of LTL, $0 < b \leq 1$ and $k \in \mathbb{N}$, if $Prob_k[\phi] \geq b$, then $Prob[\phi] \geq b$.*

Indeed, the probability of an EPF formula to be true in the bounded model of depth k is less or equal than the probability of the formula in any bounded model of depth greater than k.

This proposition can be extended to any monotone formula but we restrict our scope to EPF formulas to make our method fully automatic.

5 Approximate Probabilistic Model Checking

In order to calculate the satisfaction probability of a monotone formula, we have to verify the formula on all paths of length k. Such a computation is intractable in general since there are exponentially many paths to check. Thus, it is natural to ask: can we approximate $Prob_k[\phi]$? In this section, we propose an efficient procedure to approximate this probability. The running time of this computation is polynomial in the length of paths and the size of the formula.

In order to estimate the probabilities of monotone properties with a simple randomized algorithm, we generate random paths in the probabilistic space underlying the DTMC structure of depth k and compute a random variable A/N which estimates $Prob_k[\psi]$. To verify a statement $Prob_k[\psi] \geq b$, we test whether $A/N > b - \varepsilon$. Our decision is correct with confidence $(1 - \delta)$ after a number of samples polynomial in $\frac{1}{\varepsilon}$ and $\log \frac{1}{\delta}$. This result is obtained by using Chernoff-Hoeffding bounds [11] on the tail of the distribution of a sum of independent random variables. The main advantage of the method is that we can proceed

with just a succinct representation of the transition graph, that is a succinct description in an input language, for example Reactive Modules [1].

Definition 9. *A succinct representation, or diagram, of a PTS* $\mathcal{M} = (S, P, I, L)$ *is a representation of the PTS, that allows to generate algorithmically, for any state* s*, the set of states* t *such that* $P(s,t) > 0$*.*

The size of such a representation is in the same order of magnitude as the *PTS*. Typically, for Reactives Modules, the size of the diagram is in $\mathcal{O}(n.p)$, when the size of the *PTS* is in $\mathcal{O}(n^p)$.

In order to prove our result, we introduce the notion of fully polynomial randomized approximation scheme (FPRAS) for probability problems. This notion is analogous to randomized approximation schemes [13,19] for counting problems. Our probability problem is defined by giving as input x a succinct representation of a probabilistic system, a formula and a positive integer k. The succinct representation is used to generate a set of execution paths of length k. The solution of the probability problem is the probability measure $\mu(x)$ of the formula over the set of execution paths. The difference with randomized approximation schemes for counting problems is that for approximating probabilities, which are rational numbers in the interval $[0, 1]$, we only require approximation with additive error.

Definition 10. *A fully polynomial randomized approximation scheme (FPRAS) for a probability problem is a randomized algorithm* $\mathcal{A}$ *that takes an input* x*, two real numbers* $0 < \varepsilon, \delta < 1$ *and produces a value* $A(x, \varepsilon, \delta)$ *such that:*

$$Prob\big[|A(x,\varepsilon,\delta) - \mu(x)| \leq \varepsilon\big] \;\geq\; 1 - \delta.$$

The running time of $\mathcal{A}$ *is polynomial in* $|x|$*,* $\frac{1}{\varepsilon}$ *and* $\log \frac{1}{\delta}$*.*

The probability is taken over the random choices of the algorithm. We call ε the *approximation parameter* and δ the *confidence parameter*. By verifying the formula on $O(\frac{1}{\varepsilon^2}.\log\frac{1}{\delta})$ paths, we obtain an answer with confidence $(1-\delta)$.

Consider the following randomized algorithm designed to approximate $Prob_k[\psi]$, that is the probability of an LTL formula over bounded DTMC of depth k:

Generic approximation algorithm $\mathcal{GAA}$
Input: $diagram, \psi, k, \varepsilon, \delta$
$N := 4\log(\frac{2}{\delta})/\varepsilon^2$
$A := 0$
For $i = 1$ to N do
1. Generate a random path σ of length k with the diagram
2. If ψ is true on σ then $A := A + 1$

Return A/N

Theorem 1. *The generic approximation algorithm* $\mathcal{GAA}$ *is a fully randomized approximation scheme for the probability* $p = Prob_k[\psi]$ *for an LTL formula* ψ *and* $p \in]0, 1[$.

Proof. The random variable A is the sum of independent random variables with a Bernouilli distribution. We use the Chernoff-Hoeffding bound [11] to obtain the result. Let $X_1, ..., X_N$ be N independent random variables which take value 1 with probability p and 0 with probability $(1-p)$, and $Y = \sum_{i=1}^{N} X_i/N$. Then the Chernoff-Hoeffding bound gives $Prob\big[|Y-p| > \varepsilon\big] < 2e^{-\frac{N\varepsilon^2}{4}}$. In our case, if $N \geq 4\log(\frac{2}{\delta})/\varepsilon^2$, then $Prob\big[|A/N - p| \leq \varepsilon\big] \geq 1-\delta$ where $p = Prob_k[\psi]$.

The time needed to verify if a given path verifies ψ is polynomial in the size of the formula. The number N of iterations is polynomial in $\frac{1}{\varepsilon}$ and $\log\frac{1}{\delta}$. So $\mathcal{GAA}$ is a fully polynomial randomized approximation scheme for our probability problem.

This algorithm provides a method to verify quantitative properties expressed by EPF formulas. To check the property $Prob_k[\psi] \geq b$, we can test if the result of the approximation algorithm is greater than $b-\varepsilon$. If $Prob_k[\psi] \geq b$ is true, then the monotonicity of the property guarantees that $Prob[\psi] \geq b$ is true. Otherwise, we increment the value of k within a certain bound to conclude that $Prob[\psi] \ngeq b$.

The main problem of the method is to determine a bound on the value of k. Unfortunately, this bound might be exponential in the numbers of states, even for a simple reachability property. The bound is strongly related to the cover time of the underlying Markov chain. The problem of the computation of the cover time is known to be difficult when the input is given as a succinct representation [17].

An other way to deal with the value of k is to shrink our attention to formulas with bounded *Until* rather than classical *Until*. With this hypothesis, we can set k to the maximum time bound in some subformulas of the specification. But this is not completely satisfactory since we cannot handle general properties with only bounded until.

Now, let us discuss the parameters ε and δ. The complexity of the algorithm depends on $\log(1/\delta)$, this allows us to set δ to very small values. In our experiments, we set $\delta = 10^{-10}$, which seems to be a reasonable confidence ratio.
The dependance in ε is much more crucial, since the complexity is quadratic in $1/\varepsilon$. We set $\varepsilon = 10^{-2}$ in our experiments because this is the value that allows the best tradeoff between verification quality and time.

6 APMC: An Implementation

In this section, we present some experimental results of our approximate model checking method. These results were obtained with a tool we developed. This tool, APMC, works in a distributed framework and allows the verification of extremely large systems such as the 300 dining philosophers problem. We compare the performance of our method to the performance of PRISM. These results are promising, showing that large systems can be approximately verified in seconds, using very little memory.

APMC (Approximate Probabilistic Model Checker) is a GPL (Gnu Public License) tool written in C with lex and yacc. It uses a client/server computation

model (described in Subsection 6.2) to distribute path generation and verification on a cluster of machines.

APMC is simple to use: the user enters an LTL formula and a description of a system written in the same variant of Reactive Modules as used by PRISM. The user enters the target satisfaction probability for the property, the length of the paths to consider and the approximation and confidence parameters ε and δ. These parameters can be changed through a Graphical User Interface (GUI), represented in Figure 1. These are the basic parameters, there are advanced parameters such as the choice of a specific strategy for the speed/space compromise to use, but one can use a "by default" mode which is sufficiently efficient in general. After this, the user clicks on "go" and waits for the result. APMC is a fully automatic verification tool.

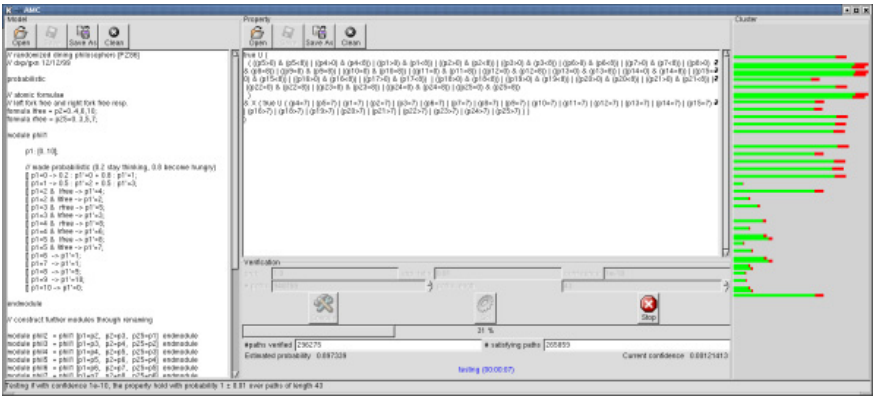


Fig. 1. The Graphical User Interface.

6.1 Standalone Use and Comparison with PRISM

We first consider a classical problem from the PRISM examples library [23]: the dining philosophers problem. Let us quickly recall the problem: n philosophers are sitting around a table, each philosopher spends most of its time thinking, but sometimes gets hungry and wants to eat. To eat, a philosopher needs both its right and left forks, but there are only n forks shared by all philosophers.
The problem is to find a protocol for the philosophers without livelock. Pnueli and Zuck [21] give a protocol that is randomized. We ran experiments on a fully probabilistic version of this protocol (that is, a DTMC version): there are no nondeterministic transitions and the scheduling between philosophers is randomized. For this protocol, we checked the following liveness property: "If a philosopher is hungry, then with probability one, some philosopher will eventually eat". This property guarantees that the protocol is livelock free. The following table shows our results using APMC and those of PRISM (model construction and model

checking time) on one 1.8 GHz Pentium 4 workstation with 512 MB of memory under the Linux operating system. For this experiment, we let $\varepsilon = 10^{-2}$ and $\delta = 10^{-10}$.

number of phil.	length	APMC (time in sec.)	PRISM (time in sec.)	PRISM (states)
3	20	35	0.394	770
5	23	56	0.87	64858
10	30	125	11.774	4.21×10^{9}
15	42	242	64.158	2.73×10^{14}
20	50	387	137.185	1.77×10^{19}
25	55	531	2469.56	1.14×10^{24}
30	65	823	out of mem.	out of mem.
50	130	3579	out of mem.	out of mem.
100	148	8364	out of mem	out of mem.

On this example, we see that we can handle larger systems than PRISM, more than 30 philosophers for Pnueli and Zuck's philosophers, without having to construct the entire model which contains 10^{24} states for 25 philosophers. Note that during the computation, our tool uses very little memory. This is due to the fact that the verification process never stores more than one path at a time.

6.2 Cluster Use

In the previous subsection, we used APMC on a single machine, but to increase the efficiency of the verification, APMC can distribute the computation on a cluster of machines using a client/server architecture.

Let us briefly describe the client/server architecture of APMC. The model, formula and other parameters are entered by the user via the Graphical User Interface which runs on the server (master). Both the model and formula are translated into C source code, compiled and sent to clients (the workers) when they request a job. Regularly, workers send current verification results, receiving an acknowledgment from the master, to know wether they have to continue or stop the computation. Since the workers only need memory to store the generated code and one path, the verification requires very little memory. Furthermore, since each path is verified independently, there is no problem of load balancing. Figure 4 shows the scalability of the implementation on Pnueli and Zuck's dining philosophers algorithm for 25 philosophers: computation time is divided by two when we double the size of the cluster. This is a consequence of very low communications overhead in the computation.

We used APMC to check properties of several fully probabilistic systems modeled as DTMCs. In Figure 2, we consider Pnueli and Zuck's Dining Philosophers algorithm [21] for which we verify the liveness property and in Figure 3, we consider a fully probabilistic version of the randomized mutual exclusion of Pnueli and Zuck [21]. All the experiments were done with a cluster of 20 workers (all are ATHLON XP1800+ under Linux) with $\varepsilon = 10^{-2}$ and $\delta = 10^{-10}$.

phil.	length	time (sec.)	max. memory (KBytes)
15	38	11	324
25	55	25	340
50	130	104	388
100	145	418	484
200	230	1399	676
300	295	4071	1012

Fig. 2. Dining philosophers: run-time and memory for 20 workers.

proc.	length	time (sec.)	max. memory (KBytes)
3	120	13	316
5	250	35	328
10	520	146	408
15	1000	882	548
20	1400	1499	660

Fig. 3. Mutual exclusion: run-time and memory for 20 workers.

We are able to verify very large systems using a reasonable cluster of workers and very little memory for each of them. In an additional experiment, with an heterogeneous cluster of 32 machines, we were able to verify the Pnueli and Zuck's 500 philosophers in about four hours.

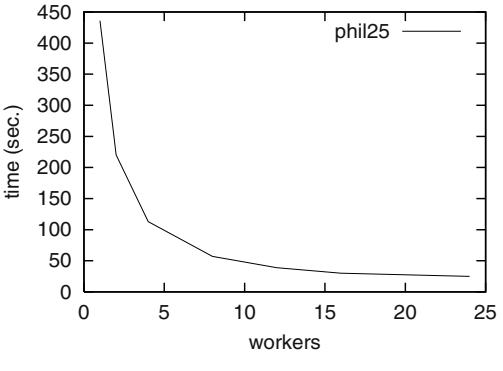


Fig. 4. Scalability of the implementation: time vs. workers for 25 dining philosophers.

7 Conclusion

To our knowledge, this work is the first to apply randomized approximation schemes to probabilistic model checking. We estimate the probability with a randomized algorithm and conclude that satisfaction probabilities of EPF formulas can be approximated. This fragment is sufficient to express reachability and livelock-freeness properties. Our implementation was used to investigate the effectiveness of this method. Our experiments point to an essential advantage of the method: the use of very little memory. In practice, this means that we are able to verify very large fully probabilistic models, such as the dining philosopher's problem with 500 philosophers. This method seems to be very useful when classical verification is intractable.

Acknowledgments. We would like to thank Sophie Laplante for many helpful discussions and suggestions, Marta Kwiatkowska and her group for their advice on PRISM. We also thank the anonymous referees for their constructive feedback.

References

1. R. Alur and T.A. Henzinger. Reactive modules. *in Proceedings of the 11th IEEE Symposium on Logic in Computer Science*, pp. 207–218, 1996.
2. APMC homepage. http://www.lri.fr/~syp/APMC
3. A. Bianco and L. de Alfaro. Model checking of probabilistic and nondeterministic systems. *Proc. FST&TCS*, Lectures Notes in Computer Science, 1026:499–513, 1995.
4. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDD's. *Proc. of 5th TACAS*, Lectures Notes in Computer Science, 1573:193–207, 1999.
5. C. Courcoubetis and M. Yannakakis. The complexity of probabilistic verification. *Journal of the ACM*, 42(4):857–907, 1995.
6. P. D'Argenio, B. Jeannet, H. Jensen and K. Larsen. Reachability analysis of probabilistic systems by successive refinements. *Proc. of the joint PAPM/PROBMIV*, 2001.
7. L. de Alfaro, M. Kwiatkowska, G. Norman, D. Parker, and R. Segala. Symbolic model checking of concurrent probabilistic processes using MTBDDs and the kronecker representation. In *Proc. of 6th TACAS*, Lectures Notes in Computer Science, 1785, 2000.
8. V. Hartonas-Garmhausen, S. Campos, and E. Clarke. Probverus: Probabilistic symbolic model checking. In *5th International AMAST Workshop, ARTS'99, May 1999*, Lecture Notes in Computer Science 1601, 1999.
9. H.Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6:512–535, 1994.
10. H. Hermanns, J.-P. Katoen, J. Meyer-Kayser and M. Siegle. A Markov chain model checker. *Proc. of 6th TACAS*, Lectures Notes in Computer Science, 1785:347–362, 2000.
11. W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58:13–30, 1963.

12. B. Jeannet, P. R. D'Argenio and K. G. Larsen RAPTURE: A tool for verifying Markov Decision Processes. In Proc. of CONCUR'02. 2002.
13. R.M. Karp, M. Luby and N. Madras. Monte-Carlo algorithms for enumeration and reliability problems. *Journal of Algorithms*, 10:429–448, 1989.
14. J. Kemeny, J. Snell and A. Knapp. *Denumerable markov chains.* Springer-Verlag, 1976.
15. M. Kwiatkowska, G. Norman and D. Parker. Probabilistic symbolic model checking with PRISM: a hybrid approach. In Proc. of 8th TACAS, Lecture Notes in Computer Science 2280, 2002.
16. R. Lassaigne and S. Peyronnet. Approximate Verification of Probabilistic Systems. In *Proc. of the 2nd joint PAPM-PROBMIV*, Lecture Notes in Computer Science 2399, 213–214, 2002.
17. L. Lovasz and P. Winkler. Exact mixing time in an unknown markov chain. *Electronic journal of combinatorics*, 1995.
18. D. Monniaux An abstract Monte-Carlo method for the analysis of probabilistic programs (extended abstract). *In 28th Symposium on Principles of Programming Languages (POPL '01)*, pages 93–101. Association for Computer Machinery, 2001.
19. R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.
20. C.H. Papadimitriou. *Computational Complexity.* Addison Wesley, 1994.
21. A. Pnueli and L. Zuck. Verification of multiprocess probabilistic protocols. *Distributed Computing*, pages 1:53–72, 1986.
22. A. Pnueli and L. Zuck. Probabilistic verification. *Information and Computation*, 103(1):1–29, 1993.
23. PRISM homepage. http://www.cs.bham.ac.uk/~dxp/prism/.
24. A. Rabinovich. Quantitative analysis of probabilistic channel systems. *proc. of 30th ICALP*, Lecture Notes in Computer Science 2719, 2003.
25. A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, 1985.
26. M.Y. Vardi. Automatic verification of probabilistic concurrent finite-state programs. *Proc. 26th Annual Symposium on Foundations of Computer Science*, pages 327–338, 1985.
27. Moshe Y. Vardi, Pierre Wolper. An Automata-Theoretic Approach to Automatic Program Verification (Preliminary Report). *In Proceedings of the first IEEE Symposium on Logic in Computer Science*, pages 332–344, 1986.
28. H. L. S. Younes and R. G. Simmons. Probabilistic Verication of Discrete Event Systems using Acceptance Sampling. *In Proc. of the 14th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science 2404, 223–235. 2002.

Completeness and Complexity of Bounded Model Checking*

Edmund Clarke[1], Daniel Kroening[1], Joël Ouaknine[1], and Ofer Strichman[2]

[1] Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA
{emc,kroening,ouaknine}@cs.cmu.edu
[2] Faculty of Industrial Engineering, Technion, Haifa, Israel
ofers@ie.technion.ac.il

Abstract. For every finite model M and an LTL property φ, there exists a number $\mathcal{CT}$ (the *Completeness Threshold*) such that if there is no counterexample to φ in M of length $\mathcal{CT}$ or less, then $M \models \varphi$. Finding this number, if it is sufficiently small, offers a practical method for making Bounded Model Checking complete. We describe how to compute an over-approximation to $\mathcal{CT}$ for a general LTL property using Büchi automata, following the Vardi-Wolper LTL model checking framework. Based on the value of $\mathcal{CT}$, we prove that the complexity of standard SAT-based BMC is doubly exponential, and that consequently there is a complexity gap of an exponent between this procedure and standard LTL model checking. We discuss ways to bridge this gap.
The article mainly focuses on observations regarding bounded model checking rather than on a presentation of new techniques.

1 Introduction

Bounded Model Checking (BMC) [2,3] is a method for finding logical errors, or proving their absence, in finite-state transition systems. It is widely regarded as a complementary technique to symbolic BDD-based model checking (see [3] for a survey of experiments with BMC conducted in industry). Given a finite transition system M, an LTL formula φ and a natural number k, a BMC procedure decides whether there exists a computation in M of length k or less that violates φ. SAT-based BMC is performed by generating a propositional formula, which is satisfiable if and only if such a path exists. BMC is conducted in an iterative process, where k is incremented until either (i) an error is found, (ii) the problem becomes intractable due to the complexity of solving the corresponding SAT instance, or (iii) k reaches some pre-computed threshold, which indicates that

* This research is supported by the Semiconductor Research Corporation (SRC) under contract no. 99-TJ-684, the National Science Foundation (NSF) under grants no. CCR-9803774 and CCR-0121547, the Office of Naval Research (ONR) and the Naval Research Laboratory (NRL) under contract no. N00014-01-1-0796, and the Army Research Office (ARO) under contract no. DAAD19-01-1-0485.

B. Steffen and G. Levi (Eds.): VMCAI 2004, LNCS 2937, pp. 85–96, 2004.

M satisfies φ. We call this threshold the *Completeness Threshold*, and denote it by $\mathcal{CT}$. $\mathcal{CT}$ is any natural number that satisfies

$$M \models_{\mathcal{CT}} \varphi \rightarrow M \models \varphi$$

where $M \models_{\mathcal{CT}} \varphi$ denotes that no computation of M of length $\mathcal{CT}$ or less violates φ. Clearly, if $M \models \varphi$ then the smallest $\mathcal{CT}$ is equal to 0, and otherwise it is equal to the length of the shortest counterexample. This implies that finding the smallest $\mathcal{CT}$ is at least as hard as checking whether $M \models \varphi$. Consequently, we concentrate on computing an over-approximation to the smallest $\mathcal{CT}$ based on graph-theoretic properties of M (such as the *diameter* of the graph representing it) and an automaton representation of $\neg\varphi$. In particular, we consider all models with the same graph-theoretic properties of M as one abstract model. Thus, this computation corresponds to finding the length of the longest shortest counterexample to φ by any one of these graphs, assuming at least one of them violates φ[1]. Thus, when we say *the* value of $\mathcal{CT}$ in the rest of the paper, we refer to the value computed by this abstraction.

The value of $\mathcal{CT}$ depends on the model M, the property φ (both the structure of φ and the propositional atoms it refers to), and the exact scheme used for translating the model and property into a propositional formula. The original translation scheme of [2], which we will soon describe, is based on a k-steps *syntactic expansion* of the formula, using Pnueli's expansion rules for LTL [8, 12] (e.g., $\mathbf{F}p \equiv p \vee \mathbf{XF}p$). With this translation, the value of $\mathcal{CT}$ was until now known only for unnested properties such as $\mathbf{G}p$ formulas [2] and $\mathbf{F}p$ formulas [11]. Computing $\mathcal{CT}$ for general LTL formulas has so far been an open problem.

In order to solve this problem we suggest to use instead a *semantic* translation scheme, based on Büchi automata, as was suggested in [6][2]. The translation is straightforward because it follows very naturally the Vardi-Wolper LTL model checking algorithm, i.e., checking for emptiness of the product of the model M and the Büchi automaton $B_{\neg\varphi}$ representing the negation of the property ϕ. Non-emptiness of $M \times B_{\neg\varphi}$, i.e., the existence of a counterexample, is proven by exhibiting a path from an initial state to a *fair loop*. We will describe in more detail this algorithm in Section 3. Deriving from this product a propositional formula $\Omega_{\varphi}(k)$ that is satisfiable iff there exists such a path of length k or less is easy: one simply needs to conjoin the k-unwinding of the product automaton with a condition for detecting a fair loop. We will give more details about this alternative BMC translation in Section 4. For now let us just mention that due to the fact that $\Omega_{\varphi}(k)$ has the same structure regardless of the property φ, it is easy to compute $\mathcal{CT}$ based on simple graph-theoretic properties of $M \times B_{\neg\varphi}$. Furthermore, the semantic translation leads to smaller CNF formulas comparing to the syntactic translation. There are two reasons for this:

[1] If this assumption does not hold, e.g., when φ is a tautology, the smallest threshold is of course 0.

[2] The authors of [6] suggested this translation in the context of Bounded Model Checking of infinite systems, without examining the implications of this translation on completeness and complexity as we do here.

1. The semantic translation benefits from the existing algorithms for constructing compact representations of LTL formulas as Büchi automata. Such optimizations are hard to achieve with the syntactic translation. For example, the syntactic translation for $\mathbf{FF}p$ results in a larger propositional formula compared to the formula generated for $\mathbf{F}p$, although these are two equivalent formulas. Existing algorithms [14] generate in this case a Büchi automaton that corresponds to the second formula in both cases.
2. The number of variables in the formula resulting from the semantic translation is linear in k, comparing to a quadratic ratio in the syntactic translation.

This paper is mainly an exposition of observations about bounded model checking[3] rather than a presentation of new techniques. In particular, we show how to compute $\mathcal{CT}$ based on the semantic translation; prove the advantages of this translation with respect to the size of the resulting formula as mentioned above, both theoretically and through experiments; and, finally, we discuss the question of the complexity of BMC. In Section 5 we show that due to the fact that $\mathcal{CT}$ can be exponential in the number of state variables, solving the corresponding SAT instance is a doubly exponential procedure in the number of state variables and the size of the formula. This implies that there is a complexity gap of an exponent between the standard BMC technique and LTL model checking. We suggest a SAT-based procedure that closes this gap while sacrificing some of the main advantages of SAT. So far our experiments show that our procedure is not better in practice than the standard SAT-based BMC, although future improvements of this technique may change this conclusion.

2 A Translation Scheme and Its Completeness Threshold

2.1 Preliminaries

A *Kripke structure* M is a quadruple $M = (S, I, T, L)$ such that (i) S is the set of states, where states are defined by valuations to a set of Boolean variables (atomic propositions) At (ii) $I \subseteq S$ is the set of initial states (iii) $T \subseteq S \times S$ is the transition relation and (iv) $L\colon S \to 2^{(At)}$ is the labeling function. Labeling is a way to attach observations to the system: for a state $s \in S$ the set $L(s)$ contains exactly those atomic propositions that hold in s. We write $p(s)$ to denote $p \in L(s)$. The initial state I and the transition relation T are given as functions in terms of At. This kind of representation, frequently called *functional form*, can be exponentially more succinct comparing to an explicit representation of the states. This fact is important for establishing the complexity of the semantic translation, as we do in Section 4.

Propositional *Linear Temporal Logic* (LTL) formulas are defined recursively: Boolean variables are in LTL; then, if $\varphi_1, \varphi_2 \in$ LTL then so are $\mathbf{F}\varphi_1$ (Future), $\mathbf{G}\varphi_1$ (Globally), $\mathbf{X}\varphi_1$ (neXt), $\varphi_1\mathbf{U}\varphi_2$ (φ_1 Until φ_2) and $\varphi_1\mathbf{W}\varphi_2$ (φ_1 Waiting-for φ_2), $\varphi_1 \vee \varphi_2$ and $\neg\varphi_1$.

[3] Some of these observations can be considered as folk theorems in the BMC community, although none of them to the best of our knowledge was previously published.

2.2 Bounded Model Checking of LTL Properties

Given an LTL property φ, a Kripke structure M and a bound k, Bounded Model Checking is performed by generating and solving a propositional formula $\Omega_\varphi(k) : [\![M]\!]_k \wedge [\![\neg\varphi]\!]_k$ where $[\![M]\!]_k$ represents the reachable states up to step k and $[\![\neg\varphi]\!]_k$ specifies which paths of length k violate φ. The satisfiability of this conjunction implies the existence of a counterexample to φ. For example, for simple invariant properties of the form $\mathbf{G}p$ the BMC formula is

$$\Omega_\varphi(k) \doteq I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg p(s_i),$$

where the left two conjuncts represent $[\![M]\!]_k$, and the right conjunct represents $[\![\neg\varphi]\!]_k$.

There are several known methods for generating $[\![\neg\varphi]\!]_k$ [2,3,7]. These translations are based on the LTL expansion rules [8,12] (e.g., $\mathbf{F}p \equiv p \vee \mathbf{XF}p$ and $\mathbf{G}p \equiv p \wedge \mathbf{XG}p$).

In the rest of this section we consider the translation scheme of Biere et al. [3] given below. This translation distinguishes between finite and infinite paths (for the latter it formulates a path ending with a loop). For a given property, it generates both translations, and concatenates them with a disjunction.

Constructing a propositional formula that captures finite paths is simple. The formula is expanded k times according to the LTL expansion rules mentioned above, where each subformula, at each location, is represented by a new variable. For example, for the operator $\mathbf{F}$, the expansion for $i \leq k$ is $[\![\mathbf{F}\varphi]\!]_k^i := [\![\varphi]\!]_k^i \vee [\![\mathbf{F}\varphi]\!]_k^{i+1}$ and for $i > k$ $[\![\mathbf{F}\varphi]\!]_k^i :=$ FALSE. Similar rules exist for the other temporal operators and for the propositional connectives.

To capture paths ending with a loop (representing infinite paths) we need to consider the state s_l ($l \leq k$), which the last state transitions to. The translation for the operator $\mathbf{F}$ for such paths is: ${}_l[\![\mathbf{F}\varphi]\!]_k^i := {}_l[\![\varphi]\!]_k^i \vee {}_l[\![\mathbf{F}\varphi]\!]_k^{\mathrm{succ}(i)}$ where $\mathrm{succ}(i) = i + 1$ if $i < k$, and $\mathrm{succ}(i) = l$ otherwise.

Finally, in order to capture all possible loops we generate $\bigvee_{l=0}^{k}({}_lL_k \wedge {}_l[\![\varphi]\!]_k^0)$ where ${}_lL_k \doteq (s_l = s_k)$, i.e., an expression that is true iff there exists a back loop from state s_k to state s_l. Each expression of the form ${}_l[\![\varphi]\!]_k^i$ or $[\![\varphi]\!]_k^i$ is represented by a new variable. The total number of variables introduced by this translation is quadratic in k. More accurately:

Proposition 1. *The syntactic translation results in $O(k \cdot |v| + (k+1)^2 \cdot |\varphi|)$ variables, where v is the set of variables defining the states of M, i.e., $|v| = O(\log |S|)$.*

Proof. Recall the structure of the formula $\Omega_\varphi(k) : [\![M]\!]_k \wedge [\![\neg\varphi]\!]_k$. The sub-formula $[\![M]\!]_k$ adds $O(k \cdot |v|)$ variables. The sub-formula $[\![\neg\varphi]\!]_k$ adds, according to the recursive translation scheme, not more than $(k+1)^2 \cdot |\varphi|$ variables, because each expression of the form ${}_l[\![\varphi]\!]_k^l$ is a new variable, and both indices i and l range over $0 \ldots k$. Further, each subformula is unfolded separately, hence leading to the result stated above.

2.3 A Completeness Threshold for Simple Properties

There are two known results regarding the value of $\mathcal{CT}$, one for $\mathbf{G}p$ and one for $\mathbf{F}p$ formulas. Their exposition requires the following definitions.

Definition 1. *The* diameter *of a finite transition system M, denoted by $d(M)$, is the longest shortest path (defined by the number of its edges) between any two reachable states of M.*

The diameter problem can be reduced to the 'all pair shortest path' problem, and therefore be solved in time polynomial in the size of the graph. In our case, however, the graph itself is exponential in the number of variables. Alternatively, one may use the formulation of this problem as satisfiability of a Quantified Boolean Formula (QBF), as suggested in [2], and later optimized in [1,13].

Definition 2. *The* recurrence diameter *of a finite transition system M, denoted by $rd(M)$ is the longest loop-free path in M between any two reachable states.*

Finding the longest loop-free path between two states is NP-complete in the size of the graph. One way to solve it with SAT was suggested in [2]. The number of variables required by their method is proportional to the length of the longest loop-free path. Hence, the SAT instance may have an exponential number of variables, and finding a solution to this instance is doubly exponential.

We denote by $d^I(M)$ and $rd^I(M)$ the *initialized* diameter and recurrence diameter, respectively, i.e., the length of the corresponding paths when they are required to start from an initial state.

For formulas of the form $\mathbf{F}p$ (i.e., counterexamples to $\mathbf{G}\neg p$ formulas), Biere et al. suggested in [2] that $\mathcal{CT}$ is less than or equal to $d(M)$ (it was later observed by several researchers independently that in fact $d^I(M)$ is sufficient). For formulas of the form $\mathbf{G}p$ formulas (counterexamples to $\mathbf{F}\neg p$ formulas), it was shown in [11] that $\mathcal{CT}$ is equal to $rd^I(M)$. Computing $\mathcal{CT}$ for general LTL formulas, as was mentioned in the introduction, has so far been an open problem.

In the next section we review how LTL model checking can be done with Büchi automata. In Section 4 we will show how a similar method can be used for generating $\Omega_\varphi(k)$.

3 LTL Model Checking with Büchi Automata

In this section we describe how model checking of LTL formulas can be done with Büchi automata, as it was first introduced by Vardi and Wolper in [15]. A labeled Büchi automaton $M = \langle S, S_0, \delta, L, F\rangle$ is a 5-tuple where S is the set of states, $S_0 \subseteq S$ is a set of initial states, $\delta \subseteq (S \times S)$ is the transition relation, L is a labeling function mapping each state to a Boolean combination of the atomic propositions, and $F \subseteq S$ is the set of accepting states. The structure of M is similar to that of a finite-state automaton, but M is used for deciding acceptance of infinite words. Given an infinite word w, $w \in \mathcal{L}(M)$ if and only if the execution of w on M passes an infinite number of times through at least one

of the states in F. In other words, if we denote by $inf(w)$ the set of states that appear infinitely often in the path of w on M, then $inf(w) \cap F \neq \emptyset$.

Every LTL formula φ can be translated into a Büchi automaton B_φ such that B_φ accepts exactly the words (paths) that satisfy φ. There are several known techniques to translate φ to B_φ[9][4]. We do not repeat the details of this construction; rather we present several examples in Fig. 1 of such translations.

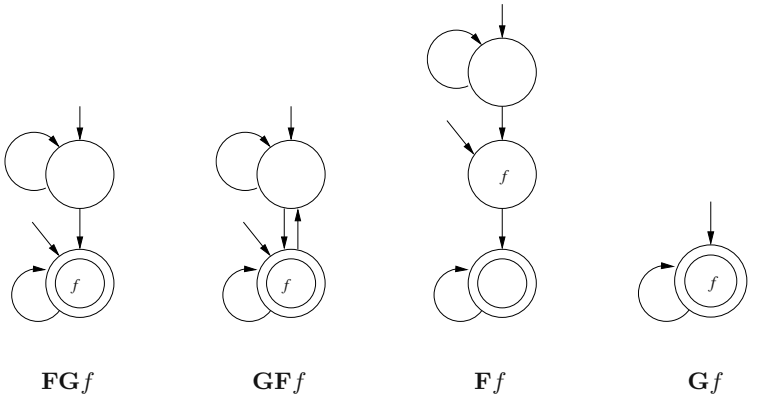


Fig. 1. Several LTL formulas and their corresponding Büchi automata. Accepting states are marked by double circles.

LTL model-checking can be done as follows: Given an LTL formula φ, construct $B_{\neg\varphi}$, a Büchi automaton that accepts exactly those paths that violate φ. Then, check whether $\Psi \doteq M \times B_{\neg\varphi}$ is empty. It is straightforward to see that $M \models \varphi$ if and only if Ψ is empty. Thus, LTL model checking is reduced to the question of Büchi automaton emptiness, i.e., proving that no word is accepted by the product automaton Ψ. In order to prove emptiness, one has to show that no computation of Ψ passes through an accepting state an infinite number of times. Consequently, finding a reachable loop in Ψ that contains an accepting state is necessary and sufficient for proving that the relation $M \not\models \varphi$ holds. Such loops are called *fair loops*.

4 The Semantic Translation

The fact that emptiness of Ψ is proven by finding a path to a fair loop gives us a straightforward adaptation of the LTL model checking procedure to a SAT-based

[4] Most published techniques for this translation construct a *generalized* Büchi automaton, while in this article we use a standard Büchi automaton (the only difference being that the former allows multiple accepting sets). The translation from generalized to standard Büchi automaton multiplies the size of the automaton by up to a factor of $|\varphi|$.

BMC procedure. This can be done by searching for a witness to the property $\varphi' \doteq \mathbf{G}(\text{TRUE})$ under the fairness constraint $\bigvee_{F_i \in F} F_i$ [5] (that is, $\bigvee_{F_i \in F} F_i$ should be true infinitely often in this path). Thus, given Ψ and k, we can use the standard BMC translation for deriving $\Omega_\Psi(k)$, a SAT instance that represents all the paths of length k that satisfy φ'. Finding such a witness of length k or less is done in BMC by solving the propositional formula:

$$\Omega_\Psi(k) \doteq I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{l=0}^{k-1} \left((s_l = s_k) \wedge \bigvee_{j=l}^{k} \bigvee_{F_i \in F} F_i(s_j) \right) \quad (1)$$

The right-most conjunct in Equation 1 constrains one of the states in F to be true in at least one of the states of the loop.

Since the Büchi automaton used in this translation captures the semantics of the property rather than its syntactic structure, we call this method a *semantic translation* for BMC.

We continue by proving the two advantages of this translation: the efficiency of the translation and the ease of computing $\mathcal{CT}$.

4.1 The Semantic Translation Is More Efficient

The semantic translation has a clear advantage in terms of the size of the resulting formula (in terms of the number of variables), as stated in the following proposition (compare to Proposition 1).

Proposition 2. *The semantic translation results in $O(k \cdot (|v| + |\varphi|))$ variables.*

Proof. The transition relation of the Büchi automaton constructed from φ is defined by $O(|\varphi|)$ variables (the automaton itself is exponential in the size of the formula, but its corresponding representation by a transition relation is as defined above). The SAT formula is constructed by unfolding k times the product Ψ, hence it uses $O(k \cdot (|v| + |\varphi|))$ variables. It also includes constraints for identifying a loop with a fair state, but these constraints only add clauses, not new variables. □

We conducted some experiments in order to check the difference between the translations. We conducted this experiment with NuSMV 2.1, which includes an optimized syntactic translation [4]. To generate the semantic translation, we derived the Büchi automaton with WRING [14] and added the resulting automaton to the NuSMV model. Then the property to be checked is simply $\mathbf{F}(\text{FALSE})$ under possible fairness constraints, as prescribed by the Büchi automaton. The table in Figure 2 summarizes these results.

As can be seen from the table and from Figure 3, there is a linear growth in the number of variables in the resulting CNF formula with the semantic translation, and a quadratic growth with the syntactic translation. Furthermore, the last three formulas have redundancy that is removed by WRING, but is not removed with the syntactic translation (observe that formulas 2 and 3 result in

Property	K	Semantic	Syntactic
$(x_0\mathbf{U}(!x_0 \wedge x_1))\mathbf{U}x_2$	7	1090	986
	15	2298	3098
	30	4563	14073
	45	6828	39898
$\mathbf{FFF}x_2$	7	528	569
	15	1112	1321
	30	2207	3076
	45	3302	5281
$\mathbf{FFFFFF}x_2$	7	528	632
	15	1112	Timeout
	30	2207	
	45	3302	
$\mathbf{GFGF}x_2$	7	586	590
	15	1234	1426
	30	2449	3511
	45	3664	Timeout

Fig. 2. The number of variables in the CNF formula resulting from the semantic and syntactic translation. The former grows linearly with k, while the latter may grow quadratically. The Timeout entry indicates that it takes NuSMV more than 15 minutes to generate the CNF formula.

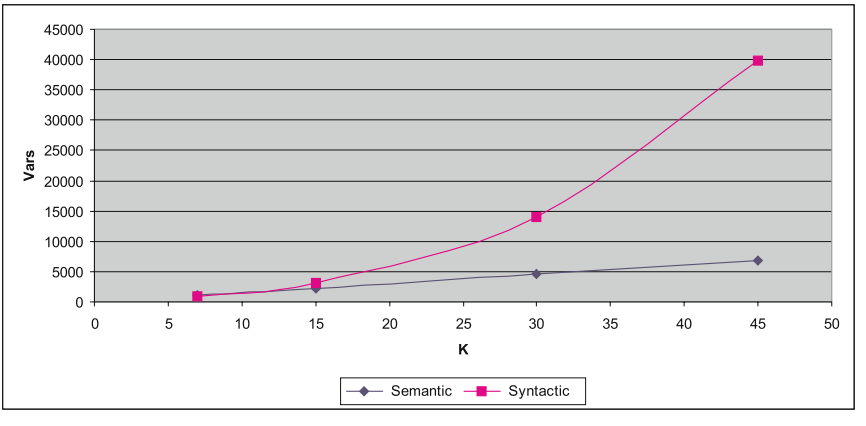


Fig. 3. The number of variables for the formula $(x_0\mathbf{U}(!x_0 \wedge x_1))\mathbf{U}x_2$ with the semantic and syntactic translations.

the same number of variables in the semantic translation, but not in the syntactic translation). The model which we experimented with was a toy example, and hence the resulting CNF was easy to solve with both translations. But it was sufficient for demonstrating the differences between the translations and that in some cases even generating the CNF formulas takes a long time with the syntactic translation.

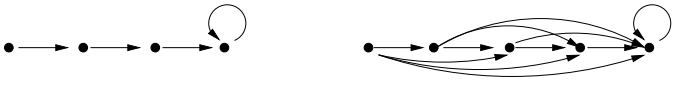
4.2 A Calculation of $\mathcal{CT}$ for LTL Based on $M \times B_{\neg\varphi}$

A major benefit of the semantic translation is that it implies directly an over-approximation of the value of $\mathcal{CT}$:

Theorem 1. *A completeness threshold for any LTL property φ when using Equation 1 is* $\min(rd^I(\Psi)+1, d^I(\Psi)+d(\Psi))$.

Proof. (a) We first prove that $\mathcal{CT}$ is bounded by $d^I(\Psi)+d(\Psi)$. If $M \not\models \varphi$ then Ψ is not empty. The shortest witness for the non-emptiness of Ψ is a path $s_0, \ldots, s_f, \ldots s_k$ where s_0 is an initial state, s_f is an accepting state and $s_k = s_l$ for some $l \leq f$. The shortest path from s_0 to s_f is not longer than $d^I(\Psi)$, and the shortest path from s_f back to itself is not longer than $d(\Psi)$. (b) We now prove that $\mathcal{CT}$ is also bounded by $rd^I(\Psi)+1$ (the addition of 1 to the longest loop-free path is needed in order to detect a loop). Falsely assume that $M \not\models \varphi$ but all witnesses are of length longer than $rd^I(\Psi)+1$. Let $W : s_0, \ldots, s_f, \ldots s_k$ be the shortest such witness. By definition of $rd^I(\Psi)$, there exists at least two states, say s_i and s_j in this path that are equal (other than the states closing the loop, i.e., $s_i \neq s_k$). If $i, j < f$ or $i, j > f$ then this path can be shortened by taking the transition from s_i to s_{j+1} (assuming, without loss of generality, that $i < j$), which contradicts our assumption that W is the shortest witness. If $i < f < j$, then the path $W' : s_0, \ldots, s_f, \ldots, s_j$ is also a loop witnessing $M \not\models \varphi$, but shorter than W, which again contradicts our assumption. □

The left-hand side drawing below demonstrates a case in which $d^I(\Psi)+d(\Psi) > rd^I(\Psi)+1$ ($d^I(\Psi) = d(\Psi) = rd^I(\Psi) = 3$), while the right-hand side drawing demonstrates the opposite case (in this case $d^I(\Psi) = d(\Psi) = 1, rd^I(\Psi)+1 = 5$). These examples justify taking the minimum between the two values.



An interesting special case is invariant properties ($\mathbf{G}p$). The Büchi automaton for the negation of this property ($\mathbf{F}\neg p$) has a special structure (see third drawing in Fig 1): for all M, any state satisfying $\neg p$ in the product $\Psi : M \times \varphi$ leads to a fair loop. Thus, to prove emptiness, it is sufficient to search for a reachable state satisfying $\neg p$. A path to such a state cannot be longer than $d^I(\Psi)$. More formally:

Theorem 2. *A completeness threshold for $\mathbf{F}p$ formulas, where p is non-temporal, is $d^I(\Psi)$.*

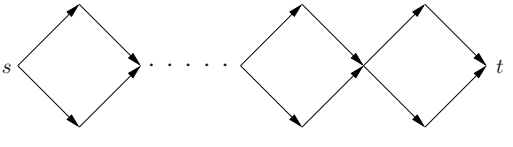
We believe that this theorem can be extended to all safety properties.

5 The Complexity of BMC

According to Theorem 1, the value of $\mathcal{CT}$ can be exponential in the number of state variables. This implies that the SAT instance (as generated in both the syntactic and semantic translations) can have an exponential number of variables and hence solving it can be doubly exponential. All known SAT-based BMC techniques, including the one presented in this article, have this complexity. Since there exists a singly exponential LTL model checking algorithm in the number of state variables, it is clear that there is a complexity gap of an exponent between the two methods. Why, then, use BMC for attempting to prove that $M \models \varphi$ holds? There are several answers to this question:

1. Indeed, BMC is normally used for detecting bugs, not for proving their absence. The number of variables in the SAT formula is polynomial in k. If the property does not hold, k depends on the location of the shallowest error. If this number is relatively small, solving the corresponding SAT instance can still be easier than competing methods.
2. In many cases the values of $rd^I(\Psi)$ and $d^I(\Psi)$ are not exponential in the number of state variables, and can even be rather small. In hardware circuits, the leading cause for exponentially long loop-free paths is counters, and hence designs without counters are much easier to solve. For example, about 25% of the components examined in [1] have a diameter smaller than 20.
3. For various technical reasons, SAT is not very sensitive to the number of variables in the formula, although theoretically it is exponential in the number of variables. Comparing it to other methods solely based on their corresponding complexity classes is not a very good indicator for their relative success in practice.

We argue that the reason for the complexity gap between SAT-based BMC and LTL model checking (as described in Section 3), is the following: SAT-based BMC does not keep track of visited states, and therefore it possibly visits the same state an exponential number of times. Unlike explicit model checking it does not explore a state graph, and unlike BDD-based symbolic model checking, it does not memorize the set of visited states. For this reason, it is possible that all paths between two states are explored, and hence a single state can be visited an exponential number of times. For example, an explicit model checking algorithm, such as the double DFS algorithm [10], will visit each state in the graph below not more than twice. SAT-based BMC, on the other hand, will consider in the worst case all 2^n possible paths between s and t, where n is the number of 'diamonds' in the graph.



A natural question is whether this complexity gap can be closed, i.e., is it possible to change the SAT-based BMC algorithm so it becomes a singly

exponential rather than a doubly exponential algorithm. Figure 4 presents a possible singly exponential BMC algorithm for $\mathbf{G}p$ formulas (i.e., reachability) based on an altered SAT algorithm that can be implemented by slightly changing a standard SAT solver. The algorithm forces the SAT solver to follow a particular variable ordering, and hence the main power of SAT (guidance of the search process with splitting heuristics) is lost. Further, it adds constrains for each visited state, forbidding the search process from revisiting it through longer or equally long paths. This potentially adds an exponential number of clauses to the formula.

1. Force a static order, following a forward traversal.
2. Each time a state i is fully evaluated (assigned):
   - Prevent the search from revisiting it through deeper paths, e.g., If $(x_i, \neg y_i)$ is a visited state, then for $i < j \leq CT$ add the following blocking state clause: $(\neg x_j \vee y_j)$.
   - When backtracking from state i, prevent the search from revisiting it in step i by adding the clause $(\neg x_i \vee y_i)$.
   - If $\neg p_i$ holds, stop and return 'Counterexample found'.

Fig. 4. A singly exponential SAT-based BMC algorithm for $\mathbf{G}p$ properties.

So far our experiments show that this procedure is worse in practice than the standard BMC[5]. Whether it is possible to find a singly exponential SAT-based algorithm that works better in practice than the standard algorithm, is still an open question with a very significant practical importance.

6 Conclusions

We discussed the advantages of the semantic translation for BMC, as was first suggested in [6]. We showed that it is in general more efficient, as it results in smaller CNF formulas, and it potentially eliminates redundancies in the property of interest. We also showed how it allows to compute the completeness threshold $\mathcal{CT}$ for all LTL formulas.

The ability to compute $\mathcal{CT}$ for general LTL enabled us to prove that all existing SAT-based BMC algorithms are doubly exponential in the number of variables. Since LTL model checking is only singly exponential in the number of variables, there is a complexity gap between the two approaches. In order to close this gap, we suggested a revised BMC algorithm that is only singly exponential, but in practice, so far, has not proved to be better than the original SAT based BMC.

[5] A. Biere implemented a similar algorithm in 2001 and reached the same conclusion. For this reason he did not publish this algorithm. Similar algorithms were also used in the past in the context of Automatic Test Pattern Generation (ATPG).

References

1. J. Baumgartner, A. Kuehlmann, and J. Abraham. Property checking via structural analysis. In E. Brinksma and K.G. Larsen, editors, *Proc.* 14^{th} *Intl. Conference on Computer Aided Verification (CAV'02)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer-Verlag.
2. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, LNCS, pages 193–207. Springer-Verlag, 1999.
3. A. Biere, A. Cimatti, E.M. Clarke, O. Strichman, and Y. Zue. *Bounded Model Checking*, volume 58 of *Advances in computers*. Academic Press, 2003.
4. Alessandro Cimatti, Marco Pistore, Marco Roveri, and Roberto Sebastiani. Improving the encoding of LTL model checking into SAT. In *VMCAI*, pages 196–207, 2002.
5. E.M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. In D. L. Dill, editor, *Proc. 6th Conference on Computer Aided Verification*, volume 818 of *Lect. Notes in Comp. Sci.*, pages 415–427. Springer-Verlag, 1994.
6. L. de Moura, H. Rueß, and M. Sorea. Lazy theorem proving for bounded model checking over infinite domains. In *Proc. 18th International Conference on Automated Deduction (CADE'02)*, Copenhagen, Denmark, July 2002.
7. A. Frisch, D. Sheridan, and T. Walsh. A fixpoint based encoding for bounded model checking. In *Formal Methods in Computer-Aided Design (FMCAD 2002)*, pages 238–255, Portland, Oregon, Nov 2002.
8. D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *Proc. 7th ACM Symp. Princ. of Prog. Lang.*, pages 163–173, 1980.
9. R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18. Chapman & Hall, 1995.
10. G.J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *Second SPIN workshop*, AMS, pages 23–32, 1996.
11. D. Kroening and O. Strichman. Efficient computation of recurrence diameters. In *4th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 2575 of *Lecture Notes in Computer Science*, pages 298–309, NYU, New-York, January 2003. Springer Verlag.
12. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1991.
13. M. Mneimneh and K. Sakallah. SAT-based sequential depth computation. In *Constraints in formal verification workshop*, Ithaca, New-York, Sep 2002.
14. F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In E. A. Emerson and A. P. Sistla, editors, *Twelfth Conference on Computer Aided Verification (CAV'00)*, pages 248–263, Berlin, July 2000. Springer-Verlag.
15. M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. First IEEE Symp. Logic in Comp. Sci.*, pages 332–344, 1986.

Model Checking for Object Specifications in Hidden Algebra

Dorel Lucanu and Gabriel Ciobanu

"A.I.Cuza" University of Iaşi, Faculty of Computer Science
{dlucanu,gabriel}@info.uaic.ro

Abstract. We use hidden algebra as a formal framework for object paradigm. We introduce a labeled transition system for each object specification model, and then define a suitable notion of bisimulation over these models. The labeled transition systems are used to define CTL models of object specifications. Given two hidden algebra models of an object specification, the bisimilar states satisfy the same set of CTL formulas. We build a canonical CTL model directly from the object specification. Using this CTL model, we can verify the temporal properties using a software tool allowing SMV model checking.

Keywords: hidden algebra, object specification, labeled transition systems, behavioral bisimulation, CTL models, model checking, SMV.

1 Introduction

Hidden algebra [4] is a theoretical framework originally aiming at giving semantics to objects. It generalizes algebraic specification by making a clear separation between visible and hidden types, the first staying for data while the second for object states. Hidden algebra has been successfully used to formally specify, simulate, and verify behavioral correctness of object systems. One of its distinctive features is the relation of *behavioral equivalence*, also known as "indistinguishability under experiments", which states that two (states of) objects are behaviorally equivalent as far as they cannot be distinguished by any of the experiments that one can or is interested to perform on the system.

The aim of the paper is to provide an automatic method to verify the temporal properties for systems specified in hidden algebra. We use CTL to express the temporal properties , and we define the satisfaction relation using labeled transition systems derived from hidden models. We present a way to verify these properties working directly over the specifications, independent of hidden models. The method can be automated using a software tool and SMV model checker.

The paper is organized in the following way. In section 2 we shortly present hidden algebra, specification of objects in hidden algebra, and the BOBJ system. We show how the mutual exclusion is described in BOBJ using the concurrent connection of object specifications. In section 3 we introduce the labeled transition system for each object model. Then we define the behavioral bisimulation

B. Steffen and G. Levi (Eds.): VMCAI 2004, LNCS 2937, pp. 97–109, 2004.

and present a nontrivial example. In section 4 we recall some basics of the Computational Tree Logic (CTL), and define the CTL models of the object specifications. We prove that bisimilar states satisfy the same set of CTL formulas. Section 5 describes a canonical CTL model of an object specification able to support model checking for temporal properties. We present briefly a procedure taking an object specification as input, and producing an SMV description of the canonical CTL model as output. In this way we extend the equational behavioral approach by adding a new dimension to the object specifications, namely the temporal requirements checkable by using our procedure and SMV. A more detailed presentation of this approach is given in [7].

2 Specification of Objects in Hidden Algebra

We assume that the reader is familiar with algebraic specification. A detailed presentation of hidden algebra can be found in [9,4]. Here we remind the main concepts and notations. A *(fixed data) hidden signature* consists of two disjoint sets: V of *visible* sorts, and H of *hidden* sorts; a many sorted $(V \cup H)$-signature Σ; and an $\Sigma{\restriction_V}$-algebra D called *data algebra*. Such a hidden signature is denoted by Σ, and its constituents are denoted by $H(\Sigma)$, $V(\Sigma)$, and $D(\Sigma)$ respectively.

Given a hidden signature (V, H, Σ, D), a *hidden Σ-model* is a Σ-algebra M such that $M{\restriction_{\Sigma{\restriction_V}}} = D$. This means that the interpretation of each sort $s \in V \cup H$ is a distinct set M_s, and the interpretation of a symbol $f \in \Sigma_{s_1 \cdots s_n, s}$ is a function $[\![f]\!]_M : M_{s_1} \times \cdots \times M_{s_n} \to M_s$. By $M{\restriction_{\Sigma{\restriction_V}}}$ we denote the algebra M restricted only to the visible sorts and visible operations. A *hidden Σ-homomorphism* $h : M \to M'$ is a Σ-homomorphism such that $h{\restriction_{\Sigma{\restriction_V}}} = \mathrm{id}_D$.

Given a hidden signature (V, H, Σ, D) and a subsignature $\Gamma \subseteq \Sigma$ such that $\Gamma{\restriction_V} = \Sigma{\restriction_V}$, a *$\Gamma$-context for sort s* is a term in $\mathcal{T}_\Gamma(\{\_ : s\} \uplus Z)$ having exactly one occurrence of a special variable $\_$ of sort s. Z is a an infinite set of distinct variables. $\mathcal{C}_\Gamma[\_ : s]$ denotes the set of all Γ-contexts for a sort s. If $c \in \mathcal{C}_\Gamma[\_ : s]$, then the sort of the term c is called the *result sort* of the context c. A Γ-context with visible result sort is called a *Γ-experiment*. If $c \in \mathcal{C}_\Gamma[\_ : s]$ with the result sort s' and $t \in \mathcal{T}_\Sigma(X)_s$, then $c[t]$ denotes the term in $\mathcal{T}_\Sigma(var(c) \cup X)$ obtained from c by substituting t for $\_$. Furthermore, for each hidden Σ-model M, c is a map $[\![c]\!]_M : M_s \to [M^{var(c)} \to M_{s'}]$ defined by $[\![c]\!]_M(a)(\vartheta) = a_\vartheta(c)$, where $a_\vartheta(c)$ is the variable assignment $\{\_ \mapsto a\} \cup \{z \mapsto \vartheta(z) \mid z \in var(c)\}$. We call $[\![c]\!]_M$ the *interpretation* of the context c in M.

Given a hidden signature (V, H, Σ, D), a subsignature $\Gamma \subseteq \Sigma$ such that $\Gamma{\restriction_V} = \Sigma{\restriction_V}$ and a hidden Σ-model M, the *Γ-behavioral equivalence* on M, denoted by $\equiv_\Sigma^\Gamma$, is defined as follows:

for any sort $s \in V \cup H$ and any $a, a' \in M_s$,

$$a \equiv_\Sigma^\Gamma a' \text{ iff } [\![c]\!]_M(a)(\vartheta) = [\![c]\!]_M(a')(\vartheta)$$

for all Γ-experiments c and all $(V \cup H)$-sorted maps $\vartheta : var(c) \to M$.

Given an equivalence $\sim$ on M, an operation $f \in \Sigma_{s_1 \ldots s_n, s}$ is *congruent wrt* $\sim$ iff $[\![f]\!]_M(a_1, \ldots, a_n) \sim [\![f]\!]_M(a'_1, \ldots, a'_n)$, whenever $a_i \sim a'_i$ for $i = \overline{1, n}$. An operation $f \in \Sigma$ is *Γ-behaviorally congruent wrt M* iff it is congruent wrt $\equiv_\Sigma^\Gamma$.

A *hidden Γ-congruence* on M is a $(V \cup H)$-equivalence which is the identity on visible sorts and each operation in Γ is congruent wrt it.

Theorem 1. *[4,9] Given a hidden signature (V, H, Σ, D), a subsignature $\Gamma \subseteq \Sigma$ such that $\Gamma{\restriction_V} = \Sigma{\restriction_V}$ and a hidden Σ-model M, then Γ-behavioral equivalence is the largest hidden Γ-congruence on M.*

A hidden Σ-model M *Γ-behaviorally satisfies* a Σ-equation e of the form $(\forall X)t = t'$ `if` C if and only if for all $\vartheta : X \to M$, $\vartheta(t) \equiv^{\Gamma}_{\Sigma} \vartheta(t')$ whenever $\vartheta(u) \equiv^{\Gamma}_{\Sigma} \vartheta(v)$ for all $u = v$ in C. We write $M \models^{\Gamma}_{\Sigma} e$. If E is a set of Σ-equations, then we write $M \models^{\Gamma}_{\Sigma} E$ iff $M \models^{\Gamma}_{\Sigma} e$ for all e in E.

An *behavioral specification* is a triplet $\mathcal{B} = (\Sigma, \Gamma, E)$ consisting of a hidden Σ-signature, a subsignature $\Gamma \subseteq \Sigma$ such that $\Gamma{\restriction_V} = \Sigma{\restriction_V}$ and a set of Σ-equations. We often denote the constituents of $\mathcal{B}$ by $\Sigma(\mathcal{B}), \Gamma(\mathcal{B})$ and respectively $E(\mathcal{B})$. The operations in $\Gamma \setminus (\Sigma{\restriction_V})$ are called *behavioral*. A hidden Σ-model M *behaviorally satisfies* the specification $\mathcal{B}$ iff M Γ-behaviorally satisfies E, that is $M \models^{\Gamma}_{\Sigma} E$. We write $M \models \mathcal{B}$ and we say that M is a *$\mathcal{B}$-model*. For any equation e, we write $\mathcal{B} \models e$ iff $M \models \mathcal{B}$ implies $M \models e$. An operation $f \in \Sigma$ is *behaviorally congruent* wrt $\mathcal{B}$ iff f is Γ-behaviorally congruent wrt each $\mathcal{B}$-model M.

Behavioral specifications can be used to model concurrent objects as follows. $\mathcal{B}$ specifies a *simple object* iff:
1. $H(\mathcal{B})$ has a unique element h called *state sort*;
2. each operation $f \in \Sigma \setminus \Sigma{\restriction_V}$ is either:
   – a hidden (generalized) constant which models an initial state, or
   – a *method* $f : hv_1 \cdots v_n \to h$ with $v_i \in V$, for $i = \overline{1,n}$, or
   – an *attribute* $f : hv_1 \cdots v_n \to v$ with $v \in V$ and $v_i \in V$, for $i = \overline{1,n}$.

In other words, the framework for simple objects is the monadic fixed-data hidden algebra [9]. A *concurrent connection* $\mathcal{B}_1 \| \cdots \| \mathcal{B}_n$ is defined as in [5] where the (composite) state sort is implemented as tupling. If h_i is the state sort of $\mathcal{B}_i$, then a composite state is a tuple $\langle S_1, \ldots, S_n \rangle$: `Tuple` where the state S_i is of sort h_i. Projection operations i^*, $i = \overline{1,n}$, are defined by $i^*(\langle S_1, \ldots, S_n \rangle = S_i$ together with the "tupling equation" $\langle 1^*T, \ldots, n^*T \rangle = T$ for each T of sort `Tuple`. We assume that all the specifications $\mathcal{B}_1, \ldots, \mathcal{B}_n$ share the same data algebra. By *object specification* we mean either a simple object specification, or a conservative extension of a concurrent connection of object specifications.

A similar approach of behavioral specifications is given in [6]. There the behavioral attributes are called *direct observers* and the behavioral methods are called *indirect observers*. However, there are some subtle differences between the two approaches, e.g. the definition for signature morphisms, but these differences are not essential for our approach. We have chosen hidden algebra because there is the software tool BOBJ able to execute behavioral specifications and to prove properties expressed in terms of behavioral equivalences.

BOBJ system [2] is used for behavioral specification, computation, and verification. BOBJ extends OBJ3 with support for behavioral specification and verification, and in particular, it provides circular coinductive rewriting with case analysis for conditional equations over behavioral theories. All examples in this paper are presented in BOBJ.

Example 1. Critical Region. A critical region is a section of code executed under mutual exclusion. A solution for mutual exclusion is provided by semaphores. The data algebra includes the process status values and the semaphore values:

```
dth DATA is
  sorts ProcStatus SemVal .
  ops idle blocked critical : -> ProcStatus .
  ops 0 1 : -> SemVal .
end
```

A process is specified in a rather abstract way as a simple object with just one method, that changes the status of the process, and just one attribute that returns the status:

```
bth PROCESS is
  sort Process . inc DATA .
  op setSt : Process ProcStatus -> Process .
  op getSt : Process -> ProcStatus .
  var P : Process . var S : ProcStatus .
  eq getSt(setSt(P, S)) = S .
end
```

Here we consider only two processes:

```
bth PR1 is  inc PROCESS * (sort Process to Pr1) . end
bth PR2 is  inc PROCESS * (sort Process to Pr2) . end
```

A semaphore is a protected variable whose value can be accessed and altered only by two primitive operations. A binary semaphore can take only the values 0 or 1. The module `SEM` given below describes a binary semaphore as a simple object. The two primitives are described by two methods, `setOn` and `setOff`, and the attribute `val` returns the value of the semaphore.

```
bth SEM is
  sort  Sem .  inc DATA .
  op val : Sem -> SemVal .
  ops setOn setOff : Sem -> Sem .
  var S : Sem .
  eq val(setOn(S)) = 1 .
  eq val(setOff(S)) = 0 .
end
```

We present the mutual exclusion considering a semaphore `SEM` working concurrently with the two processes `PR1` and `PR2`. We define four methods, `enter1`, `enter2`, `exit1`, and `exit2` implementing the previous procedures for each process. Since we consider only two processes, the queue may contain at most one blocked process and therefore it is useless. Testing the emptiness of the queue is equivalently with checking whether the other process is blocked.

```
bth CS is
  inc (PR1 || PR2 || SEM) * (sort Tuple to Cs) .
  op init : -> Cs .
  ops enter1 enter2 exit1 exit2 : Cs -> Cs .
  var C : Cs .
  eq enter1(C) =
     if(getSt(1*C) == idle)
       then if (val(3*C) == 1.SemVal)
               then <setSt(1*C, critical), 2*C, setOff(3*C)>
```

```
                else <setSt(1*C, blocked), 2*C, 3*C>
             fi
          else C
       fi .
    *** enter2(C) is similar to enter1(C)
    eq exit1(C) =
       if (getSt(1*C) == critical)
          then if (getSt(2*C) == blocked)
                  then <setSt(1*C, idle),
                        setSt(2*C, critical), 3*C>
                  else <setSt(1*C, idle), 2*C, setOn(3*C)>
               fi
          else C
       fi .
    *** exit2(C) is similar to exit1(C)
    eq val(3*init) = 1.SemVal .
    eq getSt(1*init) = idle .
    eq getSt(2*init) = idle .
end
```

The behavioral specification CS is a conservative extension of the concurrent connection PR1 || PR2 || SEM. The role of the four new added derived operations is to synchronize the three concurrent objects in order to achieve a specific task.

3 LTS and Bisimulation

The structure of the object specifications allows to associate a labeled transition system (LTS) to each object model. The idea is to use the behavioral methods in Γ as actions that produce transitions and the attributes in Γ as queries asking for information about a given state. If $\mathcal{B} = (\Sigma, \Gamma, E)$ is an object specification, then we write $\Gamma = Att(\Gamma) \cup Met(\Gamma)$, where $Att(\Gamma)$ denotes the set of attributes of Γ and $Met(\Gamma)$ denotes the set of methods of Γ. We make the assumptions that $\Gamma = \Sigma\restriction_V \cup Met(\Gamma) \cup Att(\Gamma)$ and $D_v \neq \emptyset$ for each visible sort v.

Definition 1. *Given an object specification $\mathcal{B} = (\Sigma, \Gamma, E)$, an* atomic Γ-query *for hidden sort h is a term of the form $q(\_, z_1, \ldots, z_n)$ with $q \in Att(\Gamma)$, $\_$ a special variable of sort h, and $z_1, \ldots, z_n$ are pairwise distinct visible variables. We denote by $A\mathcal{Q}_\Gamma[\_ : h]$ the set of all atomic Γ-queries for h. An* atomic Γ-action *for h is a term α of the form $g(\_, t_1, \ldots, t_n)$, where $g \in Met(\Gamma)$ is a method and t_i is either an atomic Γ-query, or a term in $\mathcal{T}_{\Sigma\restriction_V}(Z)$ with Z a set of visible variables, or an element in D for $i = \overline{1,n}$. We denote by $A\mathcal{A}_\Gamma[\_ : h]$ the set of all atomic Γ-actions.*

The dynamic behavior of a system is given by the sequences of states produced by an execution. Hidden algebra describes only partially this dynamics by providing means to investigate the behavior of the states under various experiments. We complete this partial approach by using the labeled transition system given by atomic actions over hidden models. The atomic actions correspond to the method calls. The presence of a query in an action models a communication between objects.

Definition 2. *Given an object specification $\mathcal{B} = (\Sigma, \Gamma, E)$ and a $\mathcal{B}$-model M, the* labeled transition system *defined by M is $LTS_\Gamma(M) = (M_h, A\mathcal{A}[\_ : h], \xrightarrow{\alpha}_M)$, where h is the state sort of $\mathcal{B}$ and the transition relation $\xrightarrow{\alpha}_M$ is given by: if α is an atomic action, then $a \xrightarrow{\alpha}_M a'$ iff there is a variable assignment $\vartheta : var(\alpha) \to D(\mathcal{B})$ such that $a' = [\![\alpha]\!]_M(a)(\vartheta)$.*

The expression $[\![\alpha]\!]_M(a)(\vartheta)$ means that the method designed by α in M is executed over the state a with the actual arguments $\vartheta(var(\alpha))$. The transition relation can be non-deterministic.

An interesting problem is the relationship between the LTSs defined by two models of the same specification. This question can be answered using an appropriate bisimulation notion.

Definition 3. *Given an object specification $\mathcal{B} = (\Sigma, \Gamma, E)$, two $\mathcal{B}$-models M and M', and a relation $R \subseteq M_h \times M'_h$ where h is the state sort of $\mathcal{B}$, we say that R is a Γ-*behavioral bisimulation *between M and M' iff:*
1. it is a strong bisimulation between $LTS_\Gamma(M)$ and $LTS_\Gamma(M')$, and
2. $a\,R\,a'$ implies $[\![q]\!]_M(a, d_1, \ldots, d_n) = [\![q]\!]_{M'}(a', d_1, \ldots, d_n)$ for each atomic Γ-query $q(\_, z_1, \ldots, z_n)$ and variable assignment $\{z_i \mapsto d_i \mid i = \overline{1,n}\}$.

The second condition in Definition 3 expresses the behavioral nature of the bisimulation relation. The addition of this constraint preserves the main properties for the behavioral bisimulations.

Proposition 1. *Given an object specification $\mathcal{B} = (\Sigma, \Gamma, E)$, then:*

1. *the Γ-behavioral equivalence is a Γ-behavioral bisimulation;*
2. *the inverse of a Γ-behavioral bisimulation is a Γ-behavioral bisimulation;*
3. *the composition, the union, and the intersection of two Γ-behavioral bisimulations are Γ-behavioral bisimulations.*

Given an object specification $\mathcal{B} = (\Sigma, \Gamma, E)$ and two $\mathcal{B}$-models M and M', then $M \approx^\Gamma_\Sigma M'$ iff there is a Γ-behavioral bisimulation R between M and M'.

Proposition 2. *Given an object specification $\mathcal{B} = (\Sigma, \Gamma, E)$, then $\approx^\Gamma_\Sigma$ is an equivalence relation.*

If $M \approx^\Gamma_\Sigma M'$, then $\approx^{\Sigma,\Gamma}_{M,M'}$ denotes the largest Γ-behavioral bisimulation between M and M' and it is defined by:

$$\approx^{\Sigma,\Gamma}_{M,M'} = \bigcup \{R \mid R \text{ a } \Gamma\text{-behavioral bisimulation between } M \text{ and } M'\}.$$

An interesting example of Γ-behavioral bisimulation is the following. Let $\mathcal{B} = (\Sigma, \Gamma, E)$ be an object specification with the state sort h. A *composite Γ-query for h* is coinductively defined as follows:

- each atomic Γ-query is a composite Γ-query for h, and the only occurrence of $\_$ is marked;
- for each composite Γ-query c, for each atomic Γ-action $g(\_, t_1, \ldots, t_n)$, $c[g(\_, t_1, \ldots, t_n)]$ is a composite Γ-query that denotes the term obtained from c by replacing the marked occurrence of the special variable $\_$ with $g(\_, t_1, \ldots, t_n)$, and the new occurrence of $\_$ as argument of g is marked.

We denote by $\mathcal{Q}_\Gamma[\_ : h]$ the set of all composite Γ-queries for h. Given two $\mathcal{B}$-models M, M', the relation $\approx^{\Sigma,\Gamma}_{M,M'} \subseteq M_h \times M'_h$ is defined as follows: $a \approx^{\Sigma,\Gamma}_{M,M'} a'$ iff for any composite Γ-query $c \in \mathcal{Q}_\Gamma[\_ : h]$ and for any $\vartheta : var(c) \to D(\mathcal{B})$, we have $[\![c]\!]_M(a)(\vartheta) = [\![c]\!]_{M'}(a')(\vartheta)$.

The definition of the relation $\approx^{\Sigma,\Gamma}_{M,M'}$ is similar to that of the behavioral equivalence. In fact, we can prove that if $M = M'$ then the behavioral bisimulation $\approx^{\Sigma,\Gamma}_{M,M}$ is the same with the behavioral equivalence over the state sort.

Theorem 2. *Given an object specification $\mathcal{B} = (\Sigma, \Gamma, E)$ with the state sort h and a $\mathcal{B}$-model M, then the Γ-behavioral equivalence is the same with the Γ-behavioral bisimulation over the sort h, i.e.*

$$a \equiv^{\Gamma}_{\Sigma} a' \text{ iff } a \approx^{\Sigma,\Gamma}_{M,M} a' \text{ for all states } a, a' \in M_h.$$

Proposition 3. *Given an object specification $\mathcal{B} = (\Sigma, \Gamma, E)$ and two $\mathcal{B}$-models M and M', $\approx^{\Sigma,\Gamma}_{M,M'}$ is a bisimulation whenever it is nonempty.*

There are cases when the relation $\approx^{\Sigma,\Gamma}_{M,M'}$ is empty (see [7]). However these cases are very rare in practice. In order to avoid them, we consider "well founded" object specifications $\mathcal{B}$ where $\approx^{\Sigma,\Gamma}_{M,M'}$ is nonempty for any two $\mathcal{B}$-models M and M'.

4 CTL Models

Computational Tree Logic (CTL) is a branching time logic, meaning that its model of time is a tree-like structure in which the future is not determined; there are different paths in the future, any one of which might be the "actual" path that is desired. We assume a fixed set `Atoms` of atomic propositions denoted by $p, q, r, \ldots$. Following [3], the CTL *formulas* are inductively defined as follows:

$$\phi ::= \mathtt{tt} \mid \mathtt{ff} \mid p \mid (\neg\phi) \mid (\phi_1 \wedge \phi_2) \mid \mathtt{EX}\,\phi \mid \mathtt{EG}\,\phi \mid \mathtt{E}\,[\phi_1\,\mathtt{U}\,\phi_2].$$

The intuitive meaning for each propositional connective is the usual one. The temporal connectives are pairs of symbols. The first is `E`, meaning "there Exists one path". The second one is `X`, `G`, or `U`, meaning "neXt state", "all future state (Globally)", and "Until", respectively. We can express other five operators, where `A` means "for All paths":

$$\mathtt{EF}\,\phi = \mathtt{E}[\mathtt{tt}\,\mathtt{U}\,\phi] \qquad \mathtt{AX}\,\phi = \neg\mathtt{EX}\,(\neg\phi)$$
$$\mathtt{AG}\,\phi = \neg\mathtt{EF}\,(\neg\phi) \qquad \mathtt{AF}\,\phi = \neg\mathtt{EG}\,(\neg\phi)$$
$$\mathtt{A}[\phi_1\,\mathtt{U}\,\phi_2] = \neg\mathtt{E}[\neg\phi_2\,\mathtt{U}\,(\neg\phi_1 \wedge \neg\phi_2)] \wedge \neg\mathtt{EG}\,\neg\phi_2$$

A *model* for CTL is a triple $\mathcal{M} = (S, \to, L)$ consisting of a set of *states* S endowed with a *transition relation* $\to$ such that for every state $s \in S$ there is a state $s' \in S$ with $s \to s'$, and a labeling function $L : S \to \mathcal{P}(\mathtt{Atoms})$.

Let $\mathcal{M} = (S, \to, L)$ be a model for CTL. The satisfaction relation $\mathcal{M}, s \models \phi$, expressing that the state $s \in S$ *satisfies* in $\mathcal{M}$ the CTL formula ϕ, is inductively defined as follows:

1. $\mathcal{M}, s \models \texttt{tt}$ and $\mathcal{M}, s \not\models \texttt{ff}$ for all $s \in S$.
2. $\mathcal{M}, s \models p$ iff $p \in L(s)$.
3. $\mathcal{M}, s \models \neg\phi$ iff $\mathcal{M}, s \not\models \phi$.
4. $\mathcal{M}, s \models \phi_1 \wedge \phi_2$ iff $\mathcal{M}, s \models \phi_1$ and $\mathcal{M}, s \models \phi_2$.
5. $\mathcal{M}, s \models \texttt{EX}\phi$ iff *for some* s_1 such that $s \to s_1$ we have $\mathcal{M}, s_1 \models \phi$.
6. $\mathcal{M}, s \models \texttt{EG}\phi$ iff *there is a path* $s = s_0 \to s_1 \to s_2 \to \dots$ such that $\mathcal{M}, s_i \models \phi$ for all $i = 0, 1, 2, \dots$.
7. $\mathcal{M}, s \models \texttt{E}[\phi_1 \,\texttt{U}\, \phi_2]$ iff *there is a path* $s = s_0 \to s_1 \to s_2 \to \dots$ which satisfies the formula $\phi_1 \,\texttt{U}\, \phi_2$, i.e. there is $i \in \{0, 1, 2, \dots\}$ such that $\mathcal{M}, s_i \models \phi_2$ and $\mathcal{M}, s_j \models \phi_1$ for all $j < i$.

We write $s \models \phi$ for $\mathcal{M}, s \models \phi$ whenever M is understood from the context. In order to define CTL formulas for object specification, it is enough to define the specific CTL atomic propositions.

Definition 4. *Given an object specification $\mathcal{B} = (\Sigma, \Gamma, E)$ with the state sort h, a* CTL atomic Γ-proposition for sort h *is an equation*

$$q(\_, d_1, \dots, d_n) = d \text{ with } q \in Att(\Gamma) \text{ and } d, d_1, \dots, d_n \in D(\mathcal{B}).$$

The intuitive meaning of an atomic Γ-proposition is that we obtain d whenever we execute a query $q(\_, d_1, \dots, d_n)$ over the current state.

Definition 5. *Given an object specification $\mathcal{B} = (\Sigma, \Gamma, E)$ with the state sort h and a $\mathcal{B}$-model M, we say that a state $a \in M_h$* satisfies *the atomic Γ-proposition $q(\_, d_1, \dots, d_n) = d$ iff $[\![q]\!]_M(a, d_1, \dots, d_n) = d$. We denote by $L_\Gamma(a)$ the set of atomic Γ-propositions satisfied by the state a.*

Example 2. Critical Region (continued). Consider
$\texttt{Att}(\Gamma) = \{\texttt{getSt}(1 * \_),\ \texttt{getSt}(2 * \_),\ \texttt{val}(3 * \_)\}$ and
$a = \langle \texttt{setSt}(1 * \texttt{init}, \texttt{critical}), 2 * \texttt{init}, \texttt{setOff}(3 * \texttt{init}) \rangle$
expressing that the first process is entered in the critical section, the second process in the initial state, and the semaphore is off. Then

$$L_\Gamma(a) = \{\texttt{getSt}(1 * \_) = \texttt{critical},\ \texttt{getSt}(2 * \_) = \texttt{idle},\ \texttt{val}(3 * \_) = 0\}.$$

The LTS associated to $\mathcal{B}$-models are used to build CTL models for $\mathcal{B}$.

Proposition 4. *Given an object specification $\mathcal{B} = (\Sigma, \Gamma, E)$ with the state sort h having at least one method in Γ, and a $\mathcal{B}$-model M, $LTS_\Gamma(M)$ together with the labeling function L_Γ form a CTL model.*

Theorem 3. *Consider an object specification $\mathcal{B} = (\Sigma, \Gamma, E)$ with the state sort h, two $\mathcal{B}$-models M and M', $a \in M_h$, and $a' \in M'_h$. If $a \approx^{\Sigma,\Gamma}_{M,M'} a'$, then a and a' satisfy the same set of CTL formulas.*

A converse result does not hold because we consider only behavioral attributes in our CTL atomic propositions. If we also consider the behavioral methods, then we can say that two states are bisimilar iff they satisfy the same

set of CTL formulas, obtaining in this way a result similar to Proposition 4.8 in [8].

Theorem 3 has an important meaning for the study of the temporal properties. Given a model M and its sets of CTL formulas, we may transfer these sets of CTL formulas to any other model M as follows: for a state a' of M', we search for a bisimilar state a of M and consider its set of CTL formulas. It is necessary to consider well founded specifications.

We note that the set E of equations plays a minimal role for proving the temporal properties. Next section reveals the essential role played by E in finding a canonical CTL model suitable for a model checking algorithm.

5 Model Checking for Object Specification

Model checking [3] is an automatic technique for verifying properties of finite-state systems. The properties are expressed using temporal logic, e.g. CTL, and the system is modeled by a state-transition graph. Then an efficient search procedure is used to check if the formulas expressing the system properties are satisfied by the corresponding transition graph.

CTL can express temporal behavioral properties of systems described by object specifications. In this section we investigate how we can associate a state-transition graph to an object specification such that the model checking procedure can be applied to verify the system. The temporal properties of a system can be expressed starting from an initial state, i.e. the state whose temporal properties we are searching for. This justifies the following definition.

Definition 6. *Given an object specification $\mathcal{B} = (\Sigma, \Gamma, E)$ with the state sort h, and a $\mathcal{B}$-model M, a state $a \in M_h$ is* initial *iff there is a (generalized) hidden constant t such that $a = [\![t]\!]_M$ ($a = [\![t]\!]_M(d_1, \ldots, d_n)$ for some $d_1, \ldots, d_n \in D$ if t is generalized). We say that M* satisfies *a CTL formula Φ and write $M \models \Phi$ iff $M, a \models \Phi$ for any initial state $a \in M_h$, where h is the state sort of $\mathcal{B}$. We say that M* satisfies *a set F of CTL formulas and we write $M \models F$ iff $M \models \Phi$ for each $\Phi \in F$. Finally, we say that $\mathcal{B}$* satisfies *a set F of CTL formulas and we write $\mathcal{B} \models F$ iff $M \models F$ for each $\mathcal{B}$-model M.*

Example 3. Critical Region (continued). A temporal specification for the critical region may include:

– there is no state such that both process execute the critical section:

$$\texttt{AG}(\neg(\texttt{getSt}(1^*\_) = \texttt{critical} \wedge \texttt{getSt}(2^*\_) = \texttt{critical}))$$

– if no process execute the critical section, then both processes are idle:

$$\texttt{AG}(\texttt{val}(3^*\_) = 1 \rightarrow (\texttt{getSt}(1^*\_) = \texttt{idle} \wedge \texttt{getSt}(2^*\_) = \texttt{idle}))$$

We are interested in finding the temporal properties of an object specification. BOBJ is not able always to check whether these properties are satisfied. If we build a model for the specification $\mathcal{B}$, then we can apply a model checking algorithm over the LTS defined by the model. The $\mathcal{B}$-models are admissible implementations and the construction of such a model is a tedious task. In this section

we propose a solution which can be successfully applied in many practical cases. We build a CTL model $KS(\mathcal{B})$ directly from the specification, and this step is done by a software tool. The temporal properties satisfied by any $\mathcal{B}$-model M are the same with those satisfied by $KS(\mathcal{B})$ whenever some assumptions are satisfied. Once we have the CTL model $KS(\mathcal{B})$, we can use the existing model checker SMV in order to check the desired temporal properties. We start by giving the construction of the CTL model $KS(\mathcal{B})$.

Assumptions. We consider only behavioral specifications $\mathcal{B} = (\Sigma, \Gamma, E)$ with the the state sort h and having the following properties:

1. If v is the result sort of an attribute $q \in Att(\Gamma)$, then $D(\mathcal{B})_v$ is finite.
2. Any ground term t of sort h is *defined*, i.e. for each atomic Γ-query $(\_, d_1, \ldots, d_n)$ with $d_i \in D$, there is $d \in D$ such that

$$\mathcal{B} \models q(t, d_1, \ldots, d_n) = d.$$

The definition for defined term is different from that used in [4] because we consider only atomic queries instead of arbitrary contexts.

An *abstract state* S for $\mathcal{B}$ is constructed as follows:
– we start with $S = \emptyset$, and
– for each $q \in Att(\Gamma)$ and appropriate $d_1, \ldots, d_n, d \in D$, we add the atomic CTL proposition $q(\_, d_1, \ldots, d_n) = d$ to S.

According to our assumptions, the set of all abstract states is finite. The transition relation over the abstract states is defined by: $S \xrightarrow{\alpha} S'$ iff for every $\mathcal{B}$-model M and $a \in M_h$, if $a \models \bigwedge_{s \in S} s$ and $a \xrightarrow{\alpha}_M a'$ then $a' \models \bigwedge_{s \in S'} s$.

Even if this definition is of semantic nature, we can use the BOBJ system to find out these transitions. Given an abstract state S and an action α, then S' is constructed according to the following steps:

1. Consider a hidden constant a of sort h. This is expressed in BOBJ by:
   `op a : -> h .`
2. For each CTL-proposition $q(\_, d_1, \ldots, d_n) = d$ in S, we define in BOBJ an equation of the form:
   `eq q(a, d1, ... ,dn) = d .`
3. For each $q \in Att(\Gamma)$ and for each appropriate $d_1, \ldots, d_n$ we can compute $q(\alpha(a), d_1, \ldots, d_n)$ using the BOBJ operational semantics:
   `red q(α(a), d1, ... ,dn) .`

We give as an example the BOBJ code that computes a transition for the critical section. Let a be the state such that the first process is entered in the critical section, the second is waiting for entering (blocked) and the semaphore value is zero. We assume that we want to compute the state obtained by the execution of the method `exit1(_)`. The BOBJ code is:

```
open .
op a : -> Cs .
eq val(3*a) = 0.SemVal .
eq getSt(1*a) = critical .
eq getSt(2*a) = blocked .
red getSt(1*exit1(a)) .
red getSt(2*exit1(a)) .
red val(3*exit1(a)) .
close
```

BOBJ system provides the following output:

```
reduce in CS : getSt(1* exit1(a))
result ProcStatus: idle
=======================================
reduce in CS : getSt(2* exit1(a))
result ProcStatus: critical
=======================================
reduce in CS : val(3* exit1(a))
result SemVal: 0
```

We see that we obtained the expected results: the first process became idle, the second is entered in the critical section, and the semaphore value remains unchanged.

Given a $\mathcal{B}$-model M, a state $a \in M_h$ is *reachable wrt* $\mathcal{B}$ iff there is a ground term t of sort h such that $[\![t]\!]_M = a$. By $RLTS_\Gamma(M)$ we denote the subsystem of $LTS_\Gamma(M)$ induced by the subset of the reachable states. The set of all reachable abstract states together with the transition relation and the labeling function $L(S)=S$ form a canonical LTS denoted by $KS(\mathcal{B})$.

Theorem 4. *Given an object specification $\mathcal{B} = (\Sigma, \Gamma, E)$ satisfying our assumptions and a $\mathcal{B}$-model M, then $KS(\mathcal{B})$ is a CTL model and for each reachable $a \in M_h$ there is an abstract state S_a in $KS(\mathcal{B})$ such that a and S_a satisfy the same set of CTL formulas.*

Corollary 1. *For any two $\mathcal{B}$-models M and M', the relation $\simeq^{\Sigma,\Gamma}_{M,M'} \subseteq M_h \times M'_h$ defined by $a \simeq^{\Sigma,\Gamma}_{M,M'} a'$ iff*

$[\![q]\!]_M(a, d_1 \dots, d_n) = [\![q]\!]_{M'}(a', d_1 \dots, d_n)$ *for any atomic Γ-query $q(\_, z_1, \dots, z_n)$ and variable assignment $\{z_i \mapsto d_i \mid i = \overline{1,n}\}$,*

is a Γ-behavioral bisimulation between $RLTS(M)$ and $RLTS(M')$.

We briefly present a procedure implemented by Mihai Daneş which produces an SMV description of the CTL model $KS(\mathcal{B})$ having as input an object specification $\mathcal{B}$. If $\mathcal{B}$ does not satisfy our assumptions then the procedure stops producing an error message. Once we have the SMV description of the CTL model, we use the SMV system to check the temporal specification of the system. The procedure works as follows:

1. Define an SMV variable for each query $q(\_, d_1, \dots, d_n)$. In this way, a state in $KS(\mathcal{B})$ is completely described by the set of the corresponding variable values.
2. Use BOBJ to compute the variables values for the initial states.
3. For each nonprocessed state, the tool uses BOBJ to compute the set of next states as above and adds the SMV description of each new computed transition to the output. This process is finite due to the assumptions we consider. If BOBJ fails to compute the next state, then the tool stops with failure (the second assumption is not satisfied).

6 Conclusion

This paper nontrivially extends the existing theory of hidden algebra:

1. It defines the notion of "bisimulation" between two models, which is important in order to say when two states behaviorally satisfy the same CTL properties. This notion generalizes the already existing notion of behavioral equivalence.
2. It gives an automatic procedure to extract a CTL model from special cases behavioral specifications. Behavioral satisfaction is a very hard problem in its generality (Π_2^0), so it is not surprising that we had to constrain the behavioral specifications from which we can extract CTL models automatically.
3. For the first time, a model checking procedure is given for behavioral specifications (in hidden logic) of object systems, by employing the SMV model checker.

In order to apply the model checking procedures, we present a construction of a canonical CTL model directly from the object specification, i.e. independent of model. These ingredients provide a formal engineering environment for object oriented paradigm covering both (algebraic) equational features, and temporal aspects.

We investigate how to weaken the constrains imposed over object specifications in order to produce canonical CTL-models. A challenging problem is to develop a framework for model checking *general* behavioral specifications able to specify infinite state systems and having multiple hidden types and methods with more than one hidden argument.

Acknowledgments. The authors would like to thank the anonymous referees for their useful remarks and comments.

References

1. M. Bidoit, R. Hennicker, and A. Kurz. Observational logic, constructor-based logic, and their duality. *Theoretical Computer Science*, 298(3), pp.471–510, 2003.
2. BOBJ home page. URL:http::/www.cs.ucsd.edu/groups/tatami/bobj/.
3. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 2000.
4. J. Goguen, and G. Malcolm. A hidden agenda. *Theoretical Computer Science* 245(1), pp.55–101, 2000.
5. J. Goguen, K. Lin, and G. Rosu. Circular Coinductive Rewriting. In *Proceedings, Automated Software Engineering '00*, IEEE Press, pp.123–131, 2000.
6. R. Hennicker, and M. Bidoit. Observational logic. In *Proc. 7th Int. Conf. Algebraic Methodology and Software Technology (AMAST'98)*, LNCS 1548, Springer, pp.263–277, 1999.
7. D. Lucanu, and G. Ciobanu. Behavioral Bisimulation and CTL Models for Hidden Algebraic Specifications. "A.I.Cuza" University of Iaşi, Faculty of Computer Science, 2003, TR 03–03, URL: `www.infoiasi.ro/~tr/tr.pl.cgi`

8. M. Rösiger. Coalgebras and Modal Logic. *Electronic Notes in Theoretical Computer Science*, 33, 2000
9. G. Roşu. *Hidden Logic*. PhD thesis, University of California at San Diego, 2000.
10. M. Wirsing. Algebraic Specification. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, Elsevier, pp.675–788, 1990.

Model Checking Polygonal Differential Inclusions Using Invariance Kernels

Gordon J. Pace[1] and Gerardo Schneider[2⋆]

[1] Department of Computer Science and AI, University of Malta.
[2] Department of Information Technology, Uppsala University, Sweden.
gordon.pace@um.edu.mt; gerardos@it.uu.se

Abstract. Polygonal hybrid systems are a subclass of planar hybrid automata which can be represented by piecewise constant differential inclusions. Here, we identify and compute an important object of such systems' phase portrait, namely *invariance kernels*. An *invariant set* is a set of initial points of trajectories which keep rotating in a cycle forever and the *invariance kernel* is the largest of such sets. We show that this kernel is a non-convex polygon and we give a non-iterative algorithm for computing the coordinates of its vertices and edges. Moreover, we present a breadth-first search algorithm for solving the reachability problem for such systems. Invariance kernels play an important role in the algorithm.

1 Introduction

A *hybrid system* is a system where both continuous and discrete behaviors interact with each other. A typical example is given by a discrete program that interacts with (controls, monitors, supervises) a continuous physical environment. In the last decade many (un)decidability results for a variety of problems concerning classes of hybrid systems have been given [ACH+95,ABDM00, BT00], [DM98,GM99,KV00]. One of the main research areas in hybrid systems is reachability analysis. Most of the proved decidability results are based on the existence of a finite and computable partition of the state space into classes of states which are equivalent with respect to reachability. This is the case for timed automata [AD94], and classes of rectangular automata [HKPV95] and hybrid automata with linear vector fields [LPY99]. For some particular classes of two-dimensional dynamical systems a *geometrical* method, which relies on the analysis of topological properties of the plane, has been developed. This approach has been proposed in [MP93]. There, it is shown that the reachability problem for two-dimensional systems with piece-wise constant derivatives (PCDs) is decidable. This result has been extended in [CV96] for planar piecewise Hamiltonian systems and in [ASY01] for polygonal hybrid systems, a class of nondeterministic systems that correspond to piecewise constant differential inclusions on the plane, see Fig. 1(a). For historical reasons we call such a system an SPDI [Sch02]. In [AMP95] it has been shown that the reachability problem for PCDs is undecidable for dimensions higher than two.

⋆ Supported by European project ADVANCE, Contract No. IST–1999–29082.

B. Steffen and G. Levi (Eds.): VMCAI 2004, LNCS 2937, pp. 110–121, 2004.

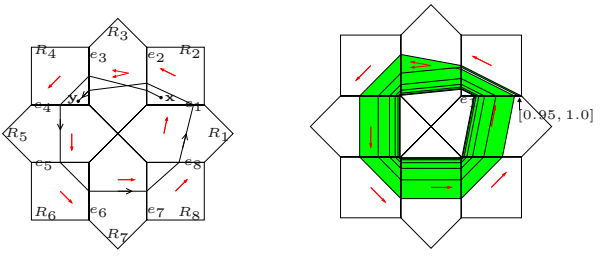


Fig. 1. (a) An SPDI and its trajectory segment; (b) Reachability analysis of the SPDI

Another important issue in the analysis of a (hybrid) dynamical system is the study of its qualitative behavior, namely the construction of its *phase portrait*. Typical questions one may want to answer include: "does every trajectory (except for the equilibrium point at the origin) converge to a limit cycle?", and "what is the biggest set such that any point on it is reachable from any other point on the set?". There are very few results on the qualitative properties of trajectories of hybrid systems [ASY02,Aub01,DV95,KV96,KdB01,MS00,SJSL00]. In particular, the question of defining and constructing phase portraits of hybrid systems has not been directly addressed except in [MS00], where phase portraits of deterministic systems with piecewise constant derivatives are explored and in [ASY02] where viability and controllability kernels for polygonal differential inclusion systems have been computed.

In this paper we show how to compute another important object of phase portraits of SPDIs, namely the *invariance kernel*. In general, an *invariant set* is a set of points such that every trajectory starting in the set remains within the set forever and the *invariance kernel* is the largest of such sets. We show that, for SPDIs, this kernel for a particular cycle is a non-convex polygon and we give a non-iterative algorithm for computing the coordinates of its vertices and edges[1]. Clearly, the invariance kernel provides useful insight about the behavior of the SPDI around simple cycles. Furthermore, we present an alternative algorithm to the one presented in [ASY01] for solving the reachability problem for SPDIs. This algorithm is a breadth-first search, in the spirit of traditional model checking algorithms. Invariance kernels play a key role in the algorithm.

2 Preliminaries

2.1 Truncated Affine Multivalued Functions

A (positive) *affine* function $f : \mathbb{R} \to \mathbb{R}$ is such that $f(x) = ax + b$ with $a > 0$. An *affine multivalued* function $F : \mathbb{R} \to 2^{\mathbb{R}}$, denoted $F = \langle f_l, f_u \rangle$, is defined by $F(x) = \langle f_l(x), f_u(x) \rangle$ where f_l and f_u are affine and $\langle \cdot, \cdot \rangle$ denotes an interval. In

[1] Notice that since SPDIs are partially defined over the plane, their invariance kernels are in general different from the whole plane.

what follows we will consider only well-formed intervals, i.e. $\langle l, u\rangle$ is an interval iff $l \leq u$. For notational convenience, we do not make explicit whether intervals are open, closed, left-open or right-open, unless required for comprehension. For an interval $I = \langle l, u\rangle$ we have that $F(\langle l, u\rangle) = \langle f_l(l), f_u(u)\rangle$. The inverse of F is defined by $F^{-1}(x) = \{y \mid x \in F(y)\}$. It is not difficult to show that $F^{-1} = \langle f_u^{-1}, f_l^{-1}\rangle$. The *universal inverse* of F is defined by $\tilde{F}^{-1}(I) = I'$ iff I' is the greatest non-empty interval such that for all $x \in I'$, $F(x) \subseteq I$. Notice that if I is a singleton then $\tilde{F}^{-1}$ is defined only if $f_l = f_u$. These classes of functions are closed under composition.

A *truncated affine multivalued* function (TAMF) $\mathcal{F} : \mathbb{R} \to 2^{\mathbb{R}}$ is defined by an affine multivalued function F and intervals $S \subseteq \mathbb{R}^+$ and $J \subseteq \mathbb{R}^+$ as follows: $\mathcal{F}(x) = F(x) \cap J$ if $x \in S$, otherwise $\mathcal{F}(x) = \emptyset$. For convenience we write $\mathcal{F}(x) = F(\{x\} \cap S) \cap J$. For an interval I, $\mathcal{F}(I) = F(I \cap S) \cap J$ and $\mathcal{F}^{-1}(I) = F^{-1}(I \cap J) \cap S$. We say that $\mathcal{F}$ is *normalized* if $S = \mathsf{Dom}\mathcal{F} = \{x \mid F(x) \cap J \neq \emptyset\}$ (thus, $S \subseteq F^{-1}(J)$) and $J = \mathsf{Im}\mathcal{F} = \mathcal{F}(S)$. In what follows we only consider normalized TAMFs. The *universal inverse* of $\mathcal{F}$ is defined by $\tilde{\mathcal{F}}^{-1}(I) = I'$ iff I' is the greatest non-empty interval such that for all $x \in I'$, $F(x) \subseteq I$ and $F(x) = \mathcal{F}(x)$.

TAMFs are closed under composition [ASY01]:

Theorem 1. The composition of two TAMFs $\mathcal{F}_1(I) = F_1(I \cap S_1) \cap J_1$ and $\mathcal{F}_2(I) = F_2(I \cap S_2) \cap J_2$, is the TAMF $(\mathcal{F}_2 \circ \mathcal{F}_1)(I) = \mathcal{F}(I) = F(I \cap S) \cap J$, where $F = F_2 \circ F_1$, $S = S_1 \cap F_1^{-1}(J_1 \cap S_2)$ and $J = J_2 \cap F_2(J_1 \cap S_2)$.

2.2 SPDI

An *angle* $\angle_{\mathbf{a}}^{\mathbf{b}}$ on the plane, defined by two non-zero vectors $\mathbf{a}, \mathbf{b}$ is the set of all positive linear combinations $\mathbf{x} = \alpha\ \mathbf{a} + \beta\ \mathbf{b}$, with $\alpha, \beta \geq 0$, and $\alpha + \beta > 0$. We can always assume that $\mathbf{b}$ is situated in the counter-clockwise direction from $\mathbf{a}$.

A *polygonal differential inclusion system* (SPDI) is defined by giving a finite partition $\mathbb{P}$ of the plane into convex polygonal sets, and associating with each $P \in \mathbb{P}$ a couple of vectors $\mathbf{a}_P$ and $\mathbf{b}_P$. Let $\phi(P) = \angle_{\mathbf{a}_P}^{\mathbf{b}_P}$. The SPDI is $\dot{\mathbf{x}} \in \phi(P)$ for $\mathbf{x} \in P$.

Let $E(P)$ be the set of edges of P. We say that $e \in E(P)$ is an *entry* of P if for all $\mathbf{x} \in e$ and for all $\mathbf{c} \in \phi(P)$, $\mathbf{x} + \mathbf{c}\epsilon \in P$ for some $\epsilon > 0$. We say that e is an *exit* of P if the same condition holds for some $\epsilon < 0$. We denote by $In(P) \subseteq E(P)$ the set of all entries of P and by $Out(P) \subseteq E(P)$ the set of all exits of P.

Assumption 1 *All the edges in $E(P)$ are either entries or exits, that is, $E(P) = In(P) \cup Out(P)$.*

Example 1. Consider the SPDI illustrated in Fig. 1(a). For each region R_i, $1 \leq i \leq 8$, there is a pair of vectors $(\mathbf{a}_i, \mathbf{b}_i)$, where: $\mathbf{a}_1 = \mathbf{b}_1 = (1, 5)$, $\mathbf{a}_2 = \mathbf{b}_2 = (-1, \frac{1}{2})$, $\mathbf{a}_3 = (-1, \frac{11}{60})$ and $\mathbf{b}_3 = (-1, -\frac{1}{10})$, $\mathbf{a}_4 = \mathbf{b}_4 = (-1, -1)$, $\mathbf{a}_5 = \mathbf{b}_5 = (0, -1)$, $\mathbf{a}_6 = \mathbf{b}_6 = (1, -1)$, $\mathbf{a}_7 = \mathbf{b}_7 = (1, 0)$, $\mathbf{a}_8 = \mathbf{b}_8 = (1, 1)$. ■

A *trajectory segment* of an SPDI is a continuous function $\xi : [0, T] \to \mathbb{R}^2$ which is smooth everywhere except in a discrete set of points, and such that for all $t \in [0, T]$, if $\xi(t) \in P$ and $\dot{\xi}(t)$ is defined then $\dot{\xi}(t) \in \phi(P)$. The *signature*, denoted $\mathsf{Sig}(\xi)$, is the ordered sequence of edges traversed by the trajectory segment, that is, $e_1, e_2, \ldots$, where $\xi(t_i) \in e_i$ and $t_i < t_{i+1}$. If $T = \infty$, a trajectory segment is called a *trajectory*.

Assumption 2 *We will only consider trajectories with infinite signatures.*

2.3 Successors and Predecessors

Given an SPDI, we fix a one-dimensional coordinate system on each edge to represent points laying on edges [ASY01]. For notational convenience, we will use e to denote both the edge and its one-dimensional representation. Accordingly, we write $\mathbf{x} \in e$ or $x \in e$, to mean "point $\mathbf{x}$ in edge e with coordinate x in the one-dimensional coordinate system of e". The same convention is applied to sets of points of e represented as intervals (e.g., $\mathbf{x} \in I$ or $x \in I$, where $I \subseteq e$) and to trajectories (e.g., "ξ starting in x" or "ξ starting in $\mathbf{x}$").
Now, let $P \in \mathbb{P}$, $e \in In(P)$ and $e' \in Out(P)$. For $I \subseteq e$, $\mathsf{Succ}_{ee'}(I)$ is the set of all points in e' reachable from some point in I by a trajectory segment $\xi : [0, t] \to \mathbb{R}^2$ in P (i.e., $\xi(0) \in I \wedge \xi(t) \in e' \wedge \mathsf{Sig}(\xi) = ee'$). We have shown in [ASY01] that $\mathsf{Succ}_{ee'}$ is a TAMF[2].

Example 2. Let $e_1, \ldots, e_8$ be as in Fig. 1(a) and $I = [l, u]$. We assume a one-dimensional coordinate system such that $e_i = S_i = J_i = (0, 1)$. We have that:

$$F_{e_1e_2}(I) = \left[\tfrac{l}{2}, \tfrac{u}{2}\right] \quad F_{e_2e_3}(I) = \left[l - \tfrac{1}{10}, u + \tfrac{11}{60}\right]$$
$$F_{e_ie_{i+1}}(I) = I \quad 3 \le i \le 7 \quad F_{e_8e_1}(I) = \left[l + \tfrac{1}{5}, u + \tfrac{1}{5}\right]$$

with $\mathsf{Succ}_{e_ie_{i+1}}(I) = F_{e_ie_{i+1}}(I \cap S_i) \cap J_{i+1}$, for $1 \le i \le 7$, and $\mathsf{Succ}_{e_8e_1}(I) = F_{e_8e_1}(I \cap S_8) \cap J_1$. ■

Given a sequence $w = e_1, e_2, \ldots, e_n$, Theorem 1 implies that the successor of I along w defined as $\mathsf{Succ}_w(I) = \mathsf{Succ}_{e_{n-1}e_n} \circ \ldots \circ \mathsf{Succ}_{e_1e_2}(I)$ is a TAMF.

Example 3. Let $\sigma = e_1 \cdots e_8e_1$. We have that $\mathsf{Succ}_\sigma(I) = F(I \cap S) \cap J$, where:

$$F(I) = \left[\tfrac{l}{2} + \tfrac{1}{10}, \tfrac{u}{2} + \tfrac{23}{60}\right]$$

$S = (0, 1)$ and $J = (\frac{1}{5}, \frac{53}{60})$ are computed using Theorem 1. ■

For $I \subseteq e'$, $\mathsf{Pre}_{ee'}(I)$ is the set of points in e that can reach a point in I by a trajectory segment in P. We have that[ASY01]: $\mathsf{Pre}_{ee'} = \mathsf{Succ}^{-1}_{ee'}$ and $\mathsf{Pre}_\sigma = \mathsf{Succ}^{-1}_\sigma$.

Example 4. Let $\sigma = e_1 \ldots e_8e_1$ be as in Fig. 1(a) and $I = [l, u]$. We have that $\mathsf{Pre}_{e_ie_{i+1}}(I) = F^{-1}_{e_ie_{i+1}}(I \cap J_{i+1}) \cap S_i$, for $1 \le i \le 7$, and $\mathsf{Pre}_{e_8e_1}(I) = F^{-1}_{e_8e_1}(I \cap J_1) \cap S_8$, where:

$$F^{-1}_{e_1e_2}(I) = [2l, 2u] \quad F^{-1}_{e_2e_3}(I) = \left[l - \tfrac{11}{60}, u + \tfrac{1}{10}\right]$$
$$F^{-1}_{e_ie_{i+1}}(I) = I \quad 3 \le i \le 7 \quad F^{-1}_{e_8e_1}(I) = \left[l - \tfrac{1}{5}, u - \tfrac{1}{5}\right]$$

Besides, $\mathsf{Pre}_\sigma(I) = F^{-1}(I \cap J) \cap S$, where $F^{-1}(I) = [2l - \frac{23}{30}, 2u - \frac{1}{5}]$. ■

[2] In [ASY01] we explain how to choose the positive direction on every edge in order to guarantee positive coefficients in the TAMF.

2.4 Qualitative Analysis of Simple Edge-Cycles

Let $\sigma = e_1 \cdots e_k e_1$ be a simple edge-cycle, i.e., $e_i \neq e_j$ for all $1 \leq i \neq j \leq k$. Let $\mathsf{Succ}_\sigma(I) = F(I \cap S) \cap J$ with $F = \langle f_l, f_u \rangle$ (we suppose that this representation is normalized). We denote by $\mathcal{D}_\sigma$ the one-dimensional discrete-time dynamical system defined by Succ_σ, that is $x_{n+1} \in \mathsf{Succ}_\sigma(x_n)$.

Assumption 3 *None of the two functions f_l, f_u is the identity.*

Let l^* and u^* be the fixpoints[3] of f_l and f_u, respectively, and $S \cap J = \langle L, U \rangle$. We have shown in [ASY01] that a simple cycle is of one of the following types:

STAY. The cycle is not abandoned neither by the leftmost nor the rightmost trajectory, that is, $L \leq l^* \leq u^* \leq U$.

DIE. The rightmost trajectory exits the cycle through the left (consequently the leftmost one also exits) or the leftmost trajectory exits the cycle through the right (consequently the rightmost one also exits), that is, $u^* < L \vee l^* > U$.

EXIT-BOTH. The leftmost trajectory exits the cycle through the left and the rightmost one through the right, that is, $l^* < L \wedge u^* > U$.

EXIT-LEFT. The leftmost trajectory exits the cycle (through the left) but the rightmost one stays inside, that is, $l^* < L \leq u^* \leq U$.

EXIT-RIGHT. The rightmost trajectory exits the cycle (through the right) but the leftmost one stays inside, that is, $L \leq l^* \leq U < u^*$.

Example 5. Let $\sigma = e_1 \cdots e_8 e_1$. We have that $S \cap J = \langle L, U \rangle = (\frac{1}{5}, \frac{53}{60})$. The fixpoints of the equation in example 3 are such that $L = l^* = \frac{1}{5} < u^* = \frac{23}{30} < U$. Thus, σ is STAY. ■

The classification above gives us some information about the qualitative behavior of trajectories. Any trajectory that enters a cycle of type DIE will eventually quit it after a finite number of turns. If the cycle is of type STAY, all trajectories that happen to enter it will keep turning inside it forever. In all other cases, some trajectories will turn for a while and then exit, and others will continue turning forever. This information is very useful for solving the reachability problem [ASY01].

Example 6. Consider again the cycle $\sigma = e_1 \cdots e_8 e_1$. Fig. 1(b) shows the reach set of the interval $[0.95, 1.0] \subset e_1$. Notice that the leftmost trajectory "converges to" the limit $l^* = \frac{1}{5}$. Fig. 1(b) has been automatically generated by the SPeeDI toolbox [APSY02] we have developed for reachability analysis of SPDIs. ■

The above result does not allow us to directly answer other questions about the behavior of the SPDI such as determine for a given point (or set of points) whether any trajectory (if it exists) starting in the point remains in the cycle forever. In order to do this, we need to further study the properties of the system around simple edge-cycles and in particular STAY cycles. See [Sch03] for some important properties of STAY cycles.

[3] Obviously, the fixpoint x^* is computed by solving a linear equation $f(x^*) = x^*$, which can be finite or infinite (see Lemma 6, page 45 of [Sch02]).

3 Invariance Kernel

In this section we define the notion of *invariance kernel* and we show how to compute it. In general, an *invariant set* is a set of points such that for any point in the set, every trajectory starting in such point remains in the set forever and the *invariance kernel* is the largest of such sets.
In particular, for SPDI, given a cyclic signature, an *invariant set* is a set of points which keep rotating in the cycle forever and the *invariance kernel* is the largest of such sets. We show that this kernel is a non-convex polygon (often with a hole in the middle) and we give a non-iterative algorithm for computing the coordinates of its vertices and edges.
In what follows, let $K \subset \mathbb{R}^2$. We recall the definition of *viable* trajectory. A trajectory ξ is *viable* in K if $\xi(t) \in K$ for all $t \geq 0$. K is a *viability domain* if for every $\mathbf{x} \in K$, there exists at least one trajectory ξ, with $\xi(0) = \mathbf{x}$, which is viable in K.

Definition 1. *We say that a set K is* invariant *if for any $x \in K$ such that there exists at least one trajectory starting in it, every trajectory starting in x is viable in K. Given a set K, its largest invariant subset is called the* invariance kernel *of K and is denoted by $\mathsf{Inv}(K_\sigma)$.* ■

We denote by $\mathcal{D}_\sigma$ the one-dimensional discrete-time dynamical system defined by Succ_σ, that is $x_{n+1} \in \mathsf{Succ}_\sigma(x_n)$. The concepts above can be defined for $\mathcal{D}_\sigma$, by setting that a trajectory $x_0 x_1 \ldots$ of $\mathcal{D}_\sigma$ is viable in an interval $I \subseteq \mathbb{R}$, if $x_i \in I$ for all $i \geq 0$. Similarly we say that an interval I in an edge e is invariant if *any* trajectory starting on $x_0 \in I$ is viable in I.
Before showing how to compute the invariance kernel of a cycle, we give a characterization of one-dimensional discrete-time invariant.

Lemma 1. *For $\mathcal{D}_\sigma$ and σ a STAY cycle, the following is valid. If I is such that $F(I) \subseteq I$ and $F(I) = \mathcal{F}(I)$ then I is invariant. On the other hand if I is invariant then $F(I) = \mathcal{F}(I)$.*

Proof: Suppose that $F(I) = \mathcal{F}(I)$ and $F(I) \subseteq I$, then $\mathcal{F}(I) \subseteq I$, thus by definition of STAY and monotonicity of $\mathcal{F}$, we know that for all n, $\mathcal{F}^n(I) \subseteq I$. Hence I is invariant. Let suppose now that I is invariant, then for any trajectory starting on $x_0 \in I$, $x_0 x_1 \ldots$ is in I and trivially $F(I) = \mathcal{F}(I)$. □
Given two edges e and e' and an interval $I \subseteq e'$ we define the $\forall$-predecessor $\widetilde{\mathsf{Pre}}(I)$ in a similar way to $\mathsf{Pre}(I)$ using the universal inverse instead of just the inverse: for $I \subseteq e'$, $\widetilde{\mathsf{Pre}}_{ee'}(I)$ is the set of points in e such that any successor of such points are in I by a trajectory segment in P. We have that $\widetilde{\mathsf{Pre}}_{ee'} = \widetilde{\mathsf{Suc}}_{ee'}^{-1}$ and $\widetilde{\mathsf{Pre}}_\sigma = \widetilde{\mathsf{Suc}}_\sigma^{-1}$.

Theorem 2. *For $\mathcal{D}_\sigma$, if $\sigma = e_1 \ldots e_n e_1$ is STAY then $\mathsf{Inv}(e_1) = \widetilde{\mathsf{Pre}}_\sigma(J)$, else $\mathsf{Inv}(e_1) = \emptyset$.*

Proof: That $\mathsf{Inv}(e_1) = \emptyset$ for any type of cycle but STAY follows directly from the definition of each type of cycle.

Let us consider a STAY cycle with signature σ. Let $I_K = \tilde{\mathcal{F}}^{-1}(J) = \widetilde{\mathsf{Pre}}_\sigma(J)$. We know that $F(\tilde{\mathcal{F}}^{-1}(J)) = \mathcal{F}(\tilde{\mathcal{F}}^{-1}(J))$[4] and by STAY property, $F(\tilde{\mathcal{F}}^{-1}(J)) \subseteq \tilde{\mathcal{F}}^{-1}(J)$, thus by Lemma 1 we have that I_K is invariant. We prove now that I_K is indeed the greatest invariant. Let suppose that there exists an invariant $H \subseteq S$ strictly greater than I_K. By assumption we have that $I_K = \tilde{\mathcal{F}}^{-1}(J) \subset H$, then by monotonicity of $\mathcal{F}$, $\mathcal{F}(\tilde{\mathcal{F}}^{-1}(J)) \subset \mathcal{F}(H)$ and since $\mathcal{F}(\tilde{\mathcal{F}}^{-1}(J)) = J$[5] we have that $J \subset \mathcal{F}(H)$, but this contradicts the monotonicity of $\mathcal{F}$ since $J = \mathcal{F}(S) \subset \mathcal{F}(H)$ and then $S \subset H$ which contradicts the hypothesis that $H \subseteq S$. Hence, $\mathsf{Inv}(e_1) = \widetilde{\mathsf{Pre}}_\sigma(J)$. □

The invariance kernel for the continuous-time system can be now found by propagating $\widetilde{\mathsf{Pre}}(J)$ from e_1 using the following operator. The *extended* $\forall$*-predecessor* of an output edge e of a region R is the set of points in R such that every trajectory segment starting in such point reaches e without traversing any other edge. More formally,

Definition 2. *Let R be a region and e be an edge in $Out(R)$. The* e-extended $\forall$-predecessor *of I, $\widetilde{\overline{\mathsf{Pre}}}_e(I)$ is defined as:*

$$\widetilde{\overline{\mathsf{Pre}}}_e(I) = \{\mathbf{x} \mid \forall \xi \,.\, (\xi(0) = \mathbf{x} \Rightarrow \exists t \geq 0 \,.\, (\xi(t) \in I \wedge \mathsf{Sig}(\xi[0,t]) = e))\}.$$ ■

The above notion can be extended to cyclic signatures (and so to edge-signatures) as follows. Let $\sigma = e_1, \ldots, e_k$ be a cyclic signature. For $I \subseteq e_1$, the σ*-extended* $\forall$*-predecessor* of I, $\widetilde{\overline{\mathsf{Pre}}}_\sigma(I)$ is the set of all $\mathbf{x} \in \mathbb{R}^2$ for which any trajectory segment ξ starting in $\mathbf{x}$, reaches some point in I, such that $\mathsf{Sig}(\xi)$ is a suffix of $e_2 \ldots e_k e_1$.

It is easy to see that $\widetilde{\overline{\mathsf{Pre}}}_\sigma(I)$ is a polygonal subset of the plane which can be calculated using the following procedure. First compute $\widetilde{\overline{\mathsf{Pre}}}_{e_i}(I)$ for all $1 \leq i \leq n$ and then apply this operation k times: $\widetilde{\overline{\mathsf{Pre}}}_\sigma(I) = \bigcup_{i=1}^{k} \widetilde{\overline{\mathsf{Pre}}}_{e_i}(I_i)$, with $I_1 = I$, $I_k = \widetilde{\mathsf{Pre}}_{e_k e_1}(I_1)$ and $I_i = \widetilde{\mathsf{Pre}}_{e_i e_{i+1}}(I_{i+1})$, for $2 \leq i \leq k-1$.

Now, let define the following set:

$$K_\sigma = \bigcup_{i=1}^{k} (\mathtt{int}(P_i) \cup e_i)$$

where P_i is such that $e_{i-1} \in In(P_i)$, $e_i \in Out(P_i)$ and $\mathtt{int}(P_i)$ is the interior of P_i.

We can now compute the invariance kernel of K_σ.

Theorem 3. *If σ is STAY then $\mathsf{Inv}(K_\sigma) = \widetilde{\overline{\mathsf{Pre}}}_\sigma(\widetilde{\mathsf{Pre}}_\sigma(J))$, otherwise $\mathsf{Inv}(K_\sigma) = \emptyset$.*

Proof: Trivially $\mathsf{Inv}(K_\sigma) = \emptyset$ for any type of cycle but STAY. That $\mathsf{Inv}(K_\sigma) = \widetilde{\overline{\mathsf{Pre}}}_\sigma(\widetilde{\mathsf{Pre}}_\sigma(J))$ for STAY cycles, follows directly from Theorem 2 and definition of $\widetilde{\overline{\mathsf{Pre}}}$. □

Example 7. Let $\sigma = e_1 \ldots e_8 e_1$. Fig. 2 depicts: (a) K_σ, and (b) $\widetilde{\overline{\mathsf{Pre}}}_\sigma(\widetilde{\mathsf{Pre}}_\sigma(J))$ ■

[4] See Lemma 13 in [Sch03] for a proof.
[5] See Lemma 12 in [Sch03] for a proof.

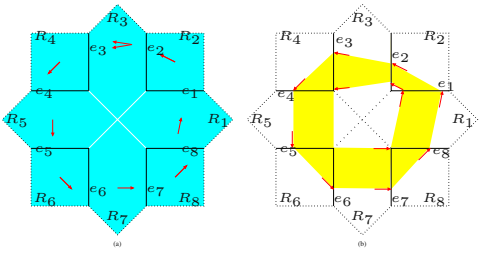


Fig. 2. Invariance kernel.

4 Reachability Algorithms for SPDIs

4.1 Previous Algorithm [ASY01]

The decidability proof of [ASY01] already provides an algorithmic way of deciding reachability in SPDIs, which was implemented in our tool SPeeDI [APSY02]. We will give an overview of the algorithm to be able to compare and contrast it with the new algorithm that we are proposing. The decidability proof is split into three steps:

1. Identify a notion of *types of signatures,* each of which 'embodies' a number of signatures through the SPDI.
2. Prove that a finite number of types suffice to cover all edge signatures. Furthermore, given an SPDI, this set is computable.
3. Give an algorithm which decides whether a given type includes a signature which is feasible under the differential inclusion constraints of the SPDI.

We will not go into the details (see [ASY01] for more details), but will outline a number of items which will allow us to compare the algorithms.

Definition 3. *A* type signature *is a sequence of edge signatures with alternating loops:* $r_1 s_1^+ r_2 s_2^+ \dots s_n^+ r_{n+1} s^*$. *The* r_i *parts of the type signature are called the* sequential paths *while the* s_i *parts called* iteration paths. *The last iteration path* s *is always a STAY loop. The interpretation of a type is similar to that of regular expressions:*
$\mathit{signatures}(r_1 s_1^+ r_2 s_2^+ \dots s_n^+ r_{n+1} s^*) \stackrel{\text{df}}{=} \{r_1 s_1^{k_1} r_2 s_2^{k_2} \dots s_n^{k_n} r_{n+1} s^k \mid k_i > 0,\ k \geq 0\}$ ■

In [ASY01], one can find details of how to decide whether a given type signature includes an edge signature which is feasible. Clearly, given a source edge e_s and a destination edge e_f, there potentially exists an infinite number of type signatures from e_s to e_f. To reduce this to a finite number, [ASY01] applies a number of syntactic constraints which ensure finiteness, but do not leave out any possibly feasible edge signatures. Using these constraints it is easy to implement a depth-first traversal of the SPDI to check all possible type signatures. Note that a breadth-first traversal would require excessive storage requirements of all intermediate nodes.

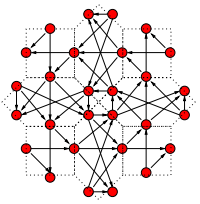


Fig. 3. The edge-graph of the swimmer SPDI example

From our experience in using SPeeDI, our implementation of this algorithm, the main deficiency of this approach is that incorrect systems which may have 'short' counter-examples (in terms of type signature length) end up lost in the exploration of long paths — either taking an excessive amount of time to find the counter-example, or coming up with a long counter-example difficult to use for debugging the hybrid system. Ideally, we should be able to find a shortest counter-example without the need of exhaustive exploration of the SPDI.

4.2 A Breadth-First Algorithm

As is evident from the previous section, it is desirable to have a breadth-first algorithm to be able to identify shortest[6] counter-examples and be able to use standard algorithms for optimisation.

Definition 4. *The* edge graph *of an SPDI with partition* $\mathbb{P}$ *is the graph with the region edges as nodes:* $N = \cup_{P \in \mathbb{P}} E(P)$*; and transitions between two edges in the same partition with the first being an input and second an output edge:* $T = \{(e, e') \mid \exists P \in \mathbb{P} \,.\, e, e' \in P,\ e \in In(P),\ e' \in Out(P)\}$. ■

Example 8. To illustrate the notion of an edge-graph, Fig. 3 illustrates the edge-graph corresponding to the SPDI representing the swimmer example given in Fig. 1(a).

Definition 5. *The* meta-graph *of an SPDI* S *is its edge graph augmented with the loops in the SPDI:*

1. *An unlabelled transition for every transition in the original graph:* $\{e \to e' \mid$ *input edge* e *and output edge* e' *belong to the same region* $\}$
2. *A set of labelled transitions, one for each simple loop in the original graph which is eventually left:* $\{e \overset{se}{\looparrowright} e' \mid \mathsf{head}(s) \neq e',\ esee' \text{ is a valid path in } S\}$
3. *A set of labelled sink edges, one for each simple loop of type STAY which is never left:* $\{e \overset{se}{\circlearrowleft} \mid ese \text{ is a valid path in } S,\ se \text{ is a STAY loop}\}$. ■

[6] Note that shortest, in this context, is not in terms or length of path on the SPDI, or number of edges visited, but on the length of the abstract signature which includes a counter-example.

Note that reachability along a path through the meta-graph corresponds to that of a type signature as defined in the previous section. For example $e_1 \rightarrow e_2 \overset{s_1}{\looparrowright} e_3 \overset{s_2}{\circlearrowleft}$ would correspond to $e_1 e_2 s_1^+ e_3 s_2^*$. From the result in [ASY01], which states that only a finite number of abstract signatures (of finite length) suffices to describe the set of all signatures through the graph, it immediately follows that for any SPDI, it suffices to explore the meta-graph to a finite depth to deduce the reachable set.

Proposition 1. *Given an SPDI S, reachability in S is equivalent to reachability in the meta-graph of S.* □

To implement the meta-graph traversal, we will define the functions corresponding to the different transitions which, given a set of edge-intervals already visited, return a new set of edge-intervals which will be visited along that transition:

$$\rightarrow(E) \overset{df}{=} \{\mathsf{Succ}_{ee'}(i) \mid i \in E,\ i \subseteq e,\ e \rightarrow e'\}$$
$$\looparrowright(E) \overset{df}{=} \{\mathsf{Succ}_p(i) \mid i \in E,\ i \subseteq e,\ e \overset{\sigma}{\looparrowright} e',\ p \text{ prefix } e\sigma^+e'\}$$
$$\circlearrowleft(E) \overset{df}{=} \{\mathsf{Succ}_p(i) \cap \mathsf{Inv}(K_\sigma) \mid i \in E,\ i \subseteq e,\ e \overset{\sigma}{\circlearrowleft},\ p \text{ prefix } e\sigma^*\}.$$

Note that, using the techniques developed in [ASY01], we can always calculate the first two of the above. Furthermore, we are guaranteed that if E consists of a finite number of edge-intervals, so will $\rightarrow(E)$ and $\looparrowright(E)$. Unfortunately, this is not the case with $\circlearrowleft(E)$. However, it is possible to compute whether a given set of edge-intervals and $\circlearrowleft(E)$ (E being a finite set of edge-intervals) overlap.

If we consider the standard model checking approach, we can use a given SPDI with transitions $\rightarrow$, meta-transitions $\looparrowright$, sink-transitions $\circlearrowleft$ and initial set I:

$$R_0 \overset{df}{=} I \qquad\qquad R_{n+1} \overset{df}{=} R_n \cup \rightarrow(R_n) \cup \looparrowright(R_n) \cup \circlearrowleft(R_n)$$

We can terminate once nothing else is added: $R_{N+1} = R_N$. Edge-interval sets R_n can be represented enumeratively. However, as already noted, STAY loops represented by sinks may induce an infinite number of disjoint intervals. However, since sinks are dead end transitions, we can simplify the reachability analysis by performing the sinks only at the end:

$$R_{n+1} \overset{df}{=} R_n \cup \rightarrow(R_n) \cup \looparrowright(R_n)$$

Since the termination condition depended on the fact that we were also applying the sink transitions $\circlearrowleft$, we add this when we check the termination condition: $R_N \cup \circlearrowleft(R_N) = R_{N+1} \cup \circlearrowleft(R_{N+1})$. Although the problem has been simply moved to the termination test, we show that this condition can be reduced to the simpler, and testable: $R_{N+1} \setminus R_N \subseteq \mathsf{Inv}$, where Inv is the set of all invariance kernels $\bigcup_{\sigma \in STAY} \mathsf{Inv}(K_\sigma)$. The proof of correctness of the algorithm can be found in [Pac03].

We can now implement the algorithm in a similar manner as standard forward model checking:

```
preR := {}; R := Src;
while (R \ preR ⊈ Inv)
    preR := R; R := R∪ → (R)∪ ↬ (R);
    if (Dst overlaps R) then return REACHABLE;
if (Dst overlaps (R ∪ ↺ (R)))
    then return REACHABLE else return UNREACHABLE;
```

5 Conclusion

One of the contributions of this paper is an automatic procedure to obtain an important object of the phase portrait of simple planar differential inclusions (SPDIs [Sch02]), namely invariance kernels.
We have also presented a breadth-first search algorithm for solving the reachability problem for SPDIs. The advantage of such an algorithm is that it is much simpler than the one presented in [ASY01] and it reminds the classical model checking algorithm for computing reachability. Invariance kernels play a crucial role to prove termination of the BFS reachability algorithm.
We intend to implement the algorithm in order to empirically compare it with the previous algorithm for SPDIs [APSY02].

Acknowledgments. We are thankful to Eugene Asarin and Sergio Yovine for the valuable discussions.

References

[ABDM00] E. Asarin, O. Bournez, T. Dang, and O. Maler. Approximate reachability analysis of piecewise-linear dynamical systems. In *Hybrid Systems: Computation and Control 2000*, LNCS 1790. Springer Verlag, 2000.

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

[AD94] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[AMP95] E. Asarin, O. Maler, and A. Pnueli. Reachability analysis of dynamical systems having piecewise-constant derivatives. *Theoretical Computer Science*, 138:35–65, 1995.

[APSY02] E. Asarin, G. Pace, G. Schneider, and S. Yovine. Speedi: a verification tool for polygonal hybrid systems. In E. Brinksma and K.G. Larsen, editors, *Computer-Aided Verification 2002*, LNCS 2404, 2002.

[ASY01] E. Asarin, G. Schneider, and S. Yovine. On the decidability of reachability for planar differential inclusions. In *Hybrid Systems: Computation and Control 2001*, LNCS 2034, 2001.

[ASY02] E. Asarin, G. Schneider, and S. Yovine. Towards computing phase portraits of polygonal differential inclusions. In *Hybrid Systems: Computation and Control 2002*, LNCS 2289, 2002.

[Aub01] J.-P. Aubin. The substratum of impulse and hybrid control systems. In *Hybrid Systems: Computation and Control 2001*, LNCS 2034, 2001.

[BT00] O. Botchkarev and S. Tripakis. Verification of hybrid systems with linear differential inclusions using ellipsoidal approximations. In *Hybrid Systems: Computation and Control 2000*, LNCS 1790, 2000.

[CV96] K. Cerāns and J. Vīksna. Deciding reachability for planar multi-polynomial systems. In *Hybrid Systems III*, LNCS 1066, 1996.

[DM98] T. Dang and O. Maler. Reachability analysis via face lifting. In *Hybrid Systems: Computation and Control 1998*, LNCS 1386, 1998.

[DV95] A. Deshpande and P. Varaiya. Viable control of hybrid systems. In *Hybrid Systems II*, LNCS 999, pages 128–147, 1995.

[GM99] M. R. Greenstreet and I. Mitchell. Reachability analysis using polygonal projections. In *HSCC '99*, LNCS 1569, 1999.

[HKPV95] T.A. Henzinger, P.W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? In *27th Annual Symposium on Theory of Computing*, pages 373–382. ACM Press, 1995.

[KdB01] P. Kowalczyk and M. di Bernardo. On a novel class of bifurcations in hybrid dynamical systems. In *Hybrid Systems: Computation and Control 2001*, LNCS 2034. Springer, 2001.

[KV96] M. Kourjanski and P. Varaiya. Stability of hybrid systems. In *Hybrid Systems III*, LNCS 1066, pages 413–423. Springer, 1996.

[KV00] A.B. Kurzhanski and P. Varaiya. Ellipsoidal techniques for reachability analysis. In *Hybrid Systems: Computation and Control*, LNCS 1790, 2000.

[LPY99] G. Lafferriere, G. J. Pappas, and S. Yovine. A new class of decidable hybrid systems. In *Hybrid Systems: Computation and Control*, LNCS 1569. 1999.

[MP93] O. Maler and A. Pnueli. Reachability analysis of planar multi-linear systems. In *Computer-Aided Verification '93*. LNCS 697, 1993.

[MS00] A. Matveev and A. Savkin. *Qualitative theory of hybrid dynamical systems*. Birkhäuser Boston, 2000.

[Pac03] G. J. Pace. A new breadth first search algorithm for deciding spdi reachability. Technical Report CSAI2003-01, Department of Computer Science & AI, University of Malta, 2003.

[Sch02] Gerardo Schneider. *Algorithmic Analysis of Polygonal Hybrid Systems*. PhD thesis, Vérimag – UJF, Grenoble, France, 2002.

[Sch03] G. Schneider. Invariance kernels of polygonal differential inclusions. Technical Report 2003-042, Department of IT, Uppsala University, 2003.

[SJSL00] S. Simić, K. Johansson, S. Sastry, and J. Lygeros. Towards a geometric theory of hybrid systems. In *Hybrid Systems: Computation and Control 2000*, LNCS 1790, 2000.

Checking Interval Based Properties for Reactive Systems*

Pei Yu[1]** and Xu Qiwen[2]

[1] International Institute for Software Technology
United Nations University, Macao SAR, P.R. China
peiyu@iist.unu.edu
[2] Faculty of Science and Technology
University of Macau, Macao SAR, P.R. China
qwxu@umac.mo

Abstract. A reactive system does not terminate and its behaviors are typically defined as a set of infinite sequences of states. In formal verification, a requirement is usually expressed in a logic, and when the models of the logic are also defined as infinite sequences, such as the case for LTL, the satisfaction relation is simply defined by the containment between the set of system behaviors and that of logic models. However, this satisfaction relation does not work for interval temporal logics, where the models can be considered as a set of finite sequences. In this paper, we observe that for different interval based properties, different satisfaction relations are sensible. Two classes of properties are discussed, and accordingly two satisfaction relations are defined, and they are subsequently unified by a more general definition. A tool is developed based on the Spin model checking system to verify the proposed general satisfaction relation for a decidable subset of Discrete Time Duration Calculus.

Keywords: model checking, finitary property, reactive system, interval temporal logic

1 Introduction

A reactive system does not terminate and its semantics is typically defined as the set of infinite sequences of states generated by the execution of the system. In formal verification, a requirement is usually expressed in a logic. A popular specification logic for reactive systems is LTL, and it is not by chance that a model of an LTL formula is also an infinite sequence of states. In this setting, the satisfaction relation is simply the containment between the set of system behaviors and that of logic models.

* This research is partly supported by University Macau Research Grant No. RG039/02-038/XQW/FST.

** On leave from Department of Computer Science and Technology, Nanjing University, P. R. China, partly supported by the National Natural Science Foundation of China (Nos. 60073031 and 60233020).

B. Steffen and G. Levi (Eds.): VMCAI 2004, LNCS 2937, pp. 122–134, 2004.

Another class of temporal logic is interval based. Although interval logics are less widely used compared to LTL, there are properties that are easier to express in such logics [14]. In almost all the interval based logics, intervals are finite ones, corresponding to finite sequences of states, which makes set containment no longer an appropriate satisfaction relation when these logics are used as the specification logic. One possible definition for this satisfaction relation is that a reactive system satisfies such a property iff all the finite prefixes of all the infinite behaviors satisfy the property, i.e. all the prefixes must be models of the requirement formula. However, while this definition makes sense in some cases, it does not in some other cases. This depends on the kind of properties. Take a mutual exclusion algorithm as an example, there are two well-known desirable properties: no two processes can be in critical sections at the same time, and a process should be allowed to enter the critical section eventually. An attempt to express these two properties in the Interval Temporal Logic (ITL) [10] or one of its variants may result in the following two formulas:

- $\Box\neg(ain \wedge bin)$: a model of it is a finite interval such that ain and bin are never true together in any of the sub-interval;
- $\Diamond ain$: a model of it is a finite interval such that ain is true in one state.

For $\Box\neg(ain \wedge bin)$, all the execution prefixes of a mutual exclusion algorithm should be its models, and if all the execution prefixes are models of the formula, intuitively the first requirement is satisfied. This is due to the fact that the property is a *safety property* [9], for which the above mentioned satisfaction relation is appropriate. On the other hand, for $\Diamond ain$, this satisfaction relation is obviously not appropriate - clearly we cannot expect for all the execution prefixes of a mutual exclusion algorithm, a particular process is in critical section. The second property is a *liveness property* [9].

There are some efforts to develop logics over infinite intervals [11,16,19]. With such logics, it is possible to use set containment as the satisfaction relation. However, logics over infinite intervals have not been widely accepted due to several reasons. Firstly, length of interval is an important concept in ITL, but for an infinite interval, this leads to an infinite number, which many people do not feel comfortable to deal with. Secondly, ITL can be decided by using finite automata when it is over finite intervals [5,12], but obviously needs automata over infinite words when the logic is defined over infinite intervals. Even worse, effective automaton constructing techniques for LTL do not seem to apply to interval logics as observed by Wolper [17].

In this paper, we propose to use the usual interval logic over finite intervals as the specification logic, and redefine the satisfaction relation. We identify two classes of properties, and accordingly define two satisfaction relations. The two relations are unified by a more general definition, and a method to automatically check whether a reactive system satisfies properties in interval logics according to this new definition is investigated. This turns out to be checking containment between the set of (infinite) behaviors of the reactive system and the language of a Büchi automaton which is the same as the finite automaton for the interval

logic formula except the final states of the finite automaton are now the accepting states of the Büchi automaton.

As a particular application of our method, we have chosen a decidable subset of Discrete Time Duration Calculus [5] as the specification logic and developed a model checking tool, integrated with the Spin model checking system, to verify the proposed general satisfaction relation.

The rest of the paper is organized as follows. In the next section, we give the syntax and the semantics of the logic, and the construction that converts a formula to a regular language representing the models of the formula. In Section 3, two kinds of properties, finitary safety property and eventual persistence property, are defined, followed by methods to recognize them. Satisfaction relations for finitary safety property and eventual persistence property are first studied separately and then unified in Section 4. A method to model check a reactive system with the proposed satisfaction relation is shown in Section 5. In Section 6, as a case study, Peterson's mutual exclusion algorithm is model checked. The paper ends with a discussion.

2 Quantified RDC

Quantified RDC(QRDC)is an extension of the decidable subset of Duration Calculus by introducing quantifiers on state variables.

We use the following notations throughout the paper. For a finite set Σ, Σ^+ denotes the set of all non-empty finite sequences over Σ and $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$, where ε is the unique empty sequence. We use Σ^ω to denote the set of infinite sequences, i.e. ω-sequences, over Σ. $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$. For a sequence α over Σ, $|\alpha|$ denotes the length of α. Specially, $|\alpha| = \infty$ for $\alpha \in \Sigma^\omega$ and $|\varepsilon| = 0$. For two sequences $\alpha_1 \in \Sigma^*$ and $\alpha_2 \in \Sigma^\infty$, $\alpha_1 \cdot \alpha_2$ denotes the sequence obtained by concatenating α_2 to the end of α_1; α_1 is called a prefix of α_2, denoted as $\alpha_1 \preceq \alpha_2$, iff $\exists \alpha_3 \in \Sigma^\infty : \alpha_1 \cdot \alpha_3 = \alpha_2$. For two sets of finite sequences P_1 and P_2, $P_1 \cdot P_2 = \{\alpha_1 \cdot \alpha_2 | \alpha_1 \in P_1 \wedge \alpha_2 \in P_2\}$.

2.1 Syntax

We use p, p_1, $\cdots$, p_n to denote state variables and $SVar$ the finite set of all the state variables. State expressions and formulas are defined as follows:

$$\begin{aligned} S &::= 0 \mid 1 \mid p \mid \neg S_1 \mid S_1 \vee S_2 \\ \phi &::= [\![S]\!] \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \phi_1; \phi_2 \mid \exists p.\ \phi \end{aligned}$$

2.2 Semantics

The interpretation for state variables is given as a function

$$I \in SVar \to (\mathbb{T}ime \to \{0, 1\})$$

When we choose $\mathbb{T}ime$ to be $\mathbb{N}$, we obtain QRDC in discrete time (DQRDC). The interpretation function I is extended to state expression S by induction on the structure of S:

1. $I[\![0]\!](t) = 0$
2. $I[\![1]\!](t) = 1$
3. $I[\![p]\!](t) = I(p)(t)$
4. $I[\![\neg S]\!](t) = 1 - I[\![S]\!](t)$
5. $I[\![S_1 \vee S_2]\!](t) = max(I[\![S_1]\!](t), I[\![S_2]\!](t))$

Given an interpretation I, the semantics of a QRDC formula ϕ is given by a function

$$I[\![\phi]\!] \in \mathbb{I}ntv \to \{tt, ff\},$$

where the set of time intervals is defined as

$$\mathbb{I}ntv \mathrel{\widehat{=}} \{[b, e] | b, e \in \mathbb{T}ime \wedge b \leqslant e\}.$$

The function is defined as follows:

1. $I[\![\lceil S \rceil]\!]([b, e]) = tt$ iff $b < e$ and $\int_b^e I[\![S]\!](t) = e - b$ for dense time or $b < e$ and $\Sigma_{t=b}^{e-1} I[\![S]\!](t) = e - b$ for discrete time
2. $I[\![\neg\phi]\!]([b, e]) = tt$ iff $I[\![\phi]\!]([b, e]) = ff$
3. $I[\![\phi_1 \vee \phi_2]\!]([b, e]) = tt$ iff $I[\![\phi_1]\!]([b, e]) = tt$ or $I[\![\phi_2]\!]([b, e]) = tt$
4. $I[\![\phi_1; \phi_2]\!]([b, e]) = tt$ iff $I[\![\phi_1]\!]([b, m]) = tt$ and $I[\![\phi_2]\!]([m, e]) = tt$, for some $m \in [b, e] \cap \mathbb{T}ime$
5. $I[\![\exists p.\ \phi]\!]([b, e]) = tt$ iff for some interpretation I', which is *p-equivalent* to I, $I'[\![\phi]\!]([b, e]) = tt$.
   An interpretation I' is p-equivalent to another interpretation I iff $I(p_1)(t) = I'(p_1)(t)$ for all $p_1 \neq p$ and $t \in \mathbb{T}ime$.

Standard abbreviations from predicate logic, e.g. "$\wedge$", "$\Rightarrow$" and "$\Leftrightarrow$", are used in this paper. Moreover, the following abbreviations for QRDC formulas are frequently used:

$[\![]\!] \mathrel{\widehat{=}} \neg[\![1]\!]$	$[\![S]\!]^* \mathrel{\widehat{=}} [\![S]\!] \vee [\![]\!]$
$false \mathrel{\widehat{=}} [\![0]\!]$	$true \mathrel{\widehat{=}} \neg false$
$\Diamond\phi \mathrel{\widehat{=}} true; \phi; true$	$\Box\phi \mathrel{\widehat{=}} \neg\Diamond\neg\phi$
$\Box_p\phi \mathrel{\widehat{=}} \neg(\neg\phi; true)$	$l = 0 \mathrel{\widehat{=}} \neg[\![1]\!]$

For DQRDC, we also use the following abbreviations($k \in \mathbb{N}$):

$l = 1 \mathrel{\widehat{=}} [\![1]\!] \wedge \neg([\![1]\!]; [\![1]\!])$	$\int S = 1 \mathrel{\widehat{=}} (\int S = 0); ([\![S]\!] \wedge l = 1); (\int S = 0)$	
$\int S = 0 \mathrel{\widehat{=}} [\![\neg S]\!] \vee l = 0$	$\int S = k + 1 \mathrel{\widehat{=}} (\int S = k); (\int S = 1)$	$(k \geqslant 1)$
$\int S \geqslant k \mathrel{\widehat{=}} (\int S = k); true$	$\int S > k \mathrel{\widehat{=}} \int S \geqslant k + 1$	
$\int S \leqslant k \mathrel{\widehat{=}} \neg(\int S > k)$	$\int S < k \mathrel{\widehat{=}} \int S \leqslant k - 1$	

where $\int S$ denotes the accumulated time when S evaluates 1. The length l of current interval can be encoded as $\int 1$.

2.3 Segments as Models

We call each $obs \in Obs = 2^{SVar}$ an observation. For a state expression S, $N(S) \subseteq Obs$ denotes the set of observations where S is evaluated to 1:

$$N(S) \mathrel{\widehat{=}} \{obs | (\bigwedge_{p_1 \in obs} p_1 \wedge \bigwedge_{p_2 \in (SVar - obs)} \neg p_2) \Rightarrow S\}.$$

For an interpretation I and $t \in \mathbb{T}ime$, $I(t)$ is identified with the following observation

$$I(t) \widehat{=} \{p | p \in SVar \wedge I(p)(t) = 1\}$$

In DQRDC, given an interpretation I and an interval $[b, e]$, we use the pair $(I, [b, e])$, called a segment, to denote the finite sequence $I(b), I(b+1), \cdots, I(e-1)$ of observations. A segment $(I, [b, e])$ satisfies a formula ϕ, denoted as $I, [b, e] \models \phi$, iff $I[\![\phi]\!]([b, e]) = tt$. For a formula ϕ, we call each segment $(I, [b, e])$ satisfying $I, [b, e] \models \phi$ a model of ϕ. The set of models of ϕ, denoted as $\mathcal{M}[\![\phi]\!]$, can be constructed [5,12] as the following regular language over the alphabet Obs:

$$\begin{array}{lll}
\mathcal{M}[\![\lceil S \rceil]\!] & = (N(S))^+ & \text{(positive closure)} \\
\mathcal{M}[\![\phi_1 \vee \phi_2]\!] & = \mathcal{M}[\![\phi_1]\!] \cup \mathcal{M}[\![\phi_2]\!] & \text{(union)} \\
\mathcal{M}[\![\neg\phi]\!] & = Obs^* - \mathcal{M}[\![\phi]\!] & \text{(complementation)} \\
\mathcal{M}[\![\phi_1; \phi_2]\!] & = \mathcal{M}[\![\phi_1]\!] \cdot \mathcal{M}[\![\phi_2]\!] & \text{(concatenation)} \\
\mathcal{M}[\![\exists p.\ \phi]\!] & = Equival_p(\mathcal{M}[\![\phi]\!]) & \text{(equivalence closure)}
\end{array}$$

where $Equival_p(\mathcal{M}[\![\phi]\!])$ is defined as follows.

Definition 1. *Given two finite sequences $\alpha = s_0, s_1, \cdots, s_n$ and $\alpha' = s'_0, s'_1, \cdots, s'_n$ of observations and a state variable p, α and α' are p-equivalent iff $s_i \backslash \{p\} = s'_i \backslash \{p\}$ for $0 \leqslant i \leqslant n$.*

Given a state variable p, the equivalence closure *of a set of observation sequences $\mathcal{M}$, $Equival_p(\mathcal{M})$ is defined to be the set*

$$\{\alpha \mid \textit{if there exists } \beta \in \mathcal{M}, \textit{ such that } \alpha \textit{ and } \beta \textit{ are } p\textit{-equivalent}\}$$

We can easily prove the following lemma by induction on the structure of ϕ.

Lemma 2. *In DQRDC, for a formula ϕ and a segment $(I, [b, e])$, $(I, [b, e]) \in \mathcal{M}[\![\phi]\!]$ iff $I, [b, e] \models \phi$.*

In DQRDC, a formula ϕ is valid iff $\mathcal{M}[\![\phi]\!] = Obs^*$, and it is satisfiable iff $\mathcal{M}[\![\phi]\!] \neq \emptyset$.

3 Finitary Safety Property and Eventual Persistence Property

Our work on classification of properties follows that of [9,3]. However, in our setting, a property is a set of finite sequences of states, and therefore some modification is necessary. To indicate whether properties are formed by finite or infinite sequences of states, we call them finitary properties or infinitary properties.

3.1 Finitary Safety Property

A safety property stipulates that "something (bad) never happens". In [3], it is considered that if a "bad" thing happens in an infinite sequence, then it must do so after some finite prefix and must be irremediable. We follow this view, but since our property is over finite sequences, we call our property *finitary safety property* and it is formally defined as follows:

Definition 3 (finitary safety property). *For a finite set Σ and a property $P \subseteq \Sigma^*$, P is a finitary safety property, or an FS property, iff:*

$$\forall \alpha \in \Sigma^* : (\alpha \notin P \Leftrightarrow \exists \beta \in \Sigma^* : (\beta \preceq \alpha \wedge \forall \gamma \in \Sigma^* : (\beta \preceq \gamma \Rightarrow \gamma \notin P))).$$

By taking negation on both sides, we have

$$\forall \alpha \in \Sigma^* : (\alpha \in P \Leftrightarrow \forall \beta \in \Sigma^* : (\beta \preceq \alpha \Rightarrow \exists \gamma \in \Sigma^* : (\beta \preceq \gamma \wedge \gamma \in P))).$$

This can be simplified, and we have the following theorem which says that a property is an FS property iff it is prefix-closed.

Theorem 4. *A non-empty property $P \subseteq \Sigma^*$ is an FS property iff*

$$\forall \alpha \in \Sigma^* : (\alpha \in P \Leftrightarrow \forall \beta \in \Sigma^* : (\beta \preceq \alpha \Rightarrow \beta \in P)),$$

i.e. $Pref(P) = P$, where $Pref(P) = \{\beta | \beta \in \Sigma^* \wedge \exists \alpha \in P : \beta \preceq \alpha\}$.

3.2 Eventual Persistence Property

An infinitary property is a liveness property [3] iff all finite executions can be extended to satisfy the property. Intuitively, this is to say a "good" thing always has the chance to take place no matter what has happened. Although in the general case, the "good thing" may take infinite number of actions to ensure, for example, something that happens infinitely often, some "good thing" can and in fact is only meaningful to occur after a finite number of steps. This is often called *eventuality*. Moreover, we are particularly interested in a class of properties such that if something ("good") has happened, nothing later can undo it. This is known as *persistence*. Formally, eventuality and persistence properties are defined as:

Definition 5 (Eventuality and Persistence). *For a finite set Σ and a property $P \subseteq \Sigma^* (P \neq \emptyset)$,*

- *P is an eventuality property iff $\forall \alpha \in \Sigma^* : \exists \beta \in \Sigma^* : (\alpha \preceq \beta \wedge \beta \in P)$;*
- *P is a persistence property iff $\forall \alpha \in P : \forall \beta \in \Sigma^* : (\alpha \preceq \beta \Rightarrow \beta \in P)$.*

P is called an eventual persistence property, or an EP property, iff P is both an eventuality property and a persistence property.

The following theorem follows directly from the definition.

Theorem 6. *A non-empty property $P \subseteq \Sigma^*$ is an EP property iff $Pref(P) = \Sigma^*$ and $P \cdot \Sigma^* = P$.*

3.3 Recognizing FS and EP Properties

A finitary property can be specified by a finite automaton whose language is the set of the finite sequences satisfying the property. In the following, similar to what has been done in [2] w.r.t. Büchi automata expressing safety properties and liveness properties, we present the rules for recognizing FS and EP properties from automata theory point of view.

A finite automaton FA is reduced if, for any state q, there is a path from an initial state to q and there is a path from q to a final state. From a finite automaton, a reduced FA accepting the same language can always be obtained by deleting those states either not reachable from any initial state or from which no final state can be reached. For a reduced FA, we define its closure $cl(FA)$ as the result automaton by setting all of its states as final states. From the construction of $cl(FA)$, we can easily get

Lemma 7. *For a reduced FA, $L(cl(FA)) = Pref(L(FA))$.*

We have from Theorem 4 and Lemma 7 the following Theorem for FS properties:

Theorem 8. *A reduced finite automaton FA specifies an FS property iff $L(FA) = L(cl(FA))$.*

For EP properties, we have the following similar theorem, which follows from Theorem 6 and Lemma 7:

Theorem 9. *A finite automaton FA specifies an EP property iff $L(cl(FA)) = \Sigma^*$ and $L(FA) \cdot \Sigma^* = L(FA)$.*

From Theorems 8 and 9, algorithms for recognizing FS and EP properties have been developed [18].

3.4 FS Formulas and EP Formulas in DQRDC

In DQRDC, each formula ϕ defines a finitary property $\mathcal{M}[\![\phi]\!]$. ϕ is called an FS/EP formula iff $\mathcal{M}[\![\phi]\!]$ is an FS/EP property. For any DQRDC formula, we can always semantically identify whether it is an FS/EP formula or not by examining the automaton accepting $\mathcal{M}[\![\phi]\!]$. The following two theorems provide syntactical conditions which are sufficient (but not necessary) to judge these two kinds of formulas. Due to limitation of space, the detailed proofs are omitted, and can be found in [18].

Theorem 10 (FS formula). *Every DQRDC formula of the form $[\![S]\!]^*$, $\int S \leqslant k (k \in \mathbb{N})$, $\Box_p \phi$ or $\Box\phi$ is an FS formula and if ϕ_1, ϕ_2 are FS formulas, then so are $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$, $\phi_1; \phi_2$ and $\exists p.\phi_1$.*

Theorem 11 (EP formula). *Every DQRDC formula of the form $\int S \geqslant k (S \neq 0$ and $k \in \mathbb{N})$ and $\Diamond\phi$ is an EP formula and if ϕ_1, ϕ_2 are EP formulas, then so are $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$, $\phi_1; \phi_2$ and $\exists p.\phi_1$.*

4 Satisfaction Relations

A finite sequence α satisfies a finitary property P, denoted by $\alpha \models P$, iff $\alpha \in P$. For an ω-sequence, we first look at the satisfaction relations for FS and EF properties separately.

For an FS property, an ω-sequence satisfies the property iff all the finite prefixes of the sequence satisfy it.

Definition 12. *For an ω-sequence σ and an FS property P all defined over Σ, σ satisfies P, denoted as $\sigma \models_i P$, iff $\forall \alpha \in \Sigma^* : (\alpha \preceq \sigma \Rightarrow \alpha \in P)$.*

Intuitively, an ω-sequence satisfies an EP property iff it has a finite prefix where the "good" thing has happened.

Definition 13. *For an ω-sequence σ and an EP property P all defined over Σ, σ satisfies P, denoted as $\sigma \models_i P$, iff $\exists \alpha \in \Sigma^* : (\alpha \preceq \sigma \wedge \alpha \in P)$.*

The two satisfaction relations can be unified by a more general one.

Definition 14 (satisfaction of finitary property). *For an ω-sequence σ and a finitary property P, both defined over Σ, σ satisfies P, denoted as $\sigma \models_i P$, iff $\exists \gamma \in \Sigma^* : (\gamma \preceq \sigma \wedge \forall \alpha \in \Sigma^* : (\gamma \preceq \alpha \preceq \sigma \Rightarrow \alpha \models P))$.*

Intuitively, this says that an ω-sequence satisfies a finitary property if all except a finite number of the finite prefixes satisfy the property.

Lemma 15. *Definition 14 is equivalent to Definition 12 for FS properties and Definition 13 for EP properties.*

Proof. For an FS property P, Theorem 4 says that P is prefix-closed, and it follows that for any ω-sequence σ over Σ, $\forall \alpha \in \Sigma^* : (\alpha \preceq \sigma \Rightarrow \alpha \in P)$ iff

$$\exists \gamma \in \Sigma^* : (\gamma \preceq \sigma \wedge \forall \alpha \in \Sigma^* : (\gamma \preceq \alpha \preceq \sigma \Rightarrow \alpha \in P)).$$

Namely, Definitions 12 and 14 are equivalent for FS properties.

For an EP property P, Definition 5 indicates that if a prefix satisfies P, then all its extensions satisfy P too. Therefore, for any ω-sequence σ over Σ, $\exists \alpha \in \Sigma^* : (\alpha \preceq \sigma \wedge \alpha \in P)$ iff

$$\exists \gamma \in \Sigma^* : (\gamma \preceq \sigma \wedge \forall \alpha \in \Sigma^* : (\gamma \preceq \alpha \preceq \sigma \Rightarrow \alpha \in P)).$$

Subsequently, Definitions 13 and 14 are equivalent for EP properties. □

5 Model Checking

Model checking [4] is a verification technique that involves constructing a model of a system and checking if a desired property holds in that model. In this section, we study model checking finitary properties specified in DQRDC.

For a system S, $\mathcal{H}[\![S]\!]$ denotes the set of all behaviors, i.e., ω-sequences of system states, of S, and S satisfies a DQRDC formula ϕ expressing finitary property iff $\forall \sigma \in \mathcal{H}[\![S]\!] : \sigma \models_i \phi$, i.e.

$$\forall \sigma \in \mathcal{H}[\![S]\!] : (\exists \gamma \in Obs^* : (\gamma \preceq \sigma \wedge \forall \alpha \in Obs^* : (\gamma \preceq \alpha \preceq \sigma \Rightarrow \alpha \models \phi))).$$

Subsequently, S does not satisfy ϕ iff

$$\exists \sigma \in \mathcal{H}[\![S]\!] : (\forall \gamma \in Obs^* : (\gamma \preceq \sigma \Rightarrow \exists \alpha \in Obs^* : (\gamma \preceq \alpha \preceq \sigma \wedge \alpha \not\models \phi))).$$

We can construct a finite automaton $FA(\neg\phi)$, which accepts precisely all the finite sequences not satisfying ϕ, thus the above condition can be rewritten as

$$\exists \sigma \in \mathcal{H}[\![S]\!] : (\forall \gamma \in Obs^* : (\gamma \preceq \sigma \Rightarrow \exists \alpha \in Obs^* : (\gamma \preceq \alpha \preceq \sigma \wedge \alpha \in L(FA(\neg\phi))))).$$

When $FA(\neg\phi)$ is deterministic, referred to as $DFA(\neg\phi)$, we shall prove this condition is equivalent to

$$\exists \sigma \in \mathcal{H}[\![S]\!] : \sigma \in L(DBA(\neg\phi)),$$

where $DBA(\neg\phi)$ is the deterministic Büchi automaton [1] obtained by taking final states of $DFA(\neg\phi)$ as accepting states.

Therefore, to determine whether S satisfies ϕ, it suffices to check whether $\mathcal{H}[\![S]\!] \cap L(DBA(\neg\phi)) = \emptyset$. If the intersection is empty, S satisfies ϕ, and if not, S does not satisfy ϕ.

Lemma 16. *For an ω-sequence σ and a deterministic finite automaton DFA, let DBA be the deterministic Büchi automaton obtained by taking all the final states of DFA as accepting states, then*
$\forall \alpha \in Obs^* : \alpha \preceq \sigma : (\exists \gamma \in Obs^* : \alpha \preceq \gamma \preceq \sigma : \alpha \in L(DFA))$ *iff* $\sigma \in L(DBA)$.

Proof. The if case is obvious and we only prove the only if case. From the assumption, there exists an infinite series of finite sequences $\alpha_1, \alpha_2, \cdots$, with $\alpha_1 \preceq \alpha_2 \preceq \cdots \preceq \sigma$, $\alpha_i \neq \alpha_{i+1}$ and $\alpha_i \in L(DFA)$ for all $i \geqslant 1$. Obviously, α_1 ends in a final state. Because the automaton is deterministic, α_2 will first go through the same states as α_1, in particular, the first final state, and then ends in a (possibly different) final state. Therefore, α_2 goes through the final states which are the accepting states of the Büchi automaton at least twice. By induction, we know that α_i goes through the accepting states of the Büchi automaton at least i times. Since there are infinite many such α_i's, some accepting states are gone through infinite number of times by σ, and therefore $\sigma \in L(DBA)$. □

Note in the above proof, the assumption that the automaton is deterministic is important. However, this is not necessary for checking FS properties. If ϕ is an FS property, in $FA(\neg\phi)$, any input symbol will lead a final state to (another) final state. Informally this is because for an FS property, if something "bad" has happened in a behavior, any extension of it is still a bad behavior. Therefore, if a prefix of σ has reached a final state, any further input will lead to a final

state, and subsequently σ is accepted by the corresponding Büchi automaton. Since determinization can be expensive, it is in general more efficient to use non-deterministic automata for model checking FS properties.

Spin [7] is a widely distributed software system for on-the-fly model checking. In Spin, the system is modelled using Promela, its input language, as a set of interacting processes. The negation of the desired property is encoded in a temporal claim, which also takes the form of a Promela program. Each process is translated into an automaton, and the global behavior of the system is obtained by computing the asynchronous product of these automata. The temporal claim is translated to a Büchi automaton. The synchronous product of this automaton and the automaton representing the global behavior of the system is computed. If the language accepted by the resulting Büchi automaton is empty, no behavior of the system satisfies the negation of the desired property, i.e. the system satisfies the property. Otherwise, the language contains some system behaviors violating the desired property.

We have implemented the algorithm which takes a DQRDC formula as input and generates the temporal claim in Promela. This allows us to model check properties studied in this paper when they are expressed as DQRDC formulas using Spin. Our implementation makes use of the C++ package for finite state machine, Grail+ [8]. The compiler construction program Parser Generator from Bumble-Bee Software Ltd. is also used.

6 Case Study

In this section, we study model checking of the well-known Peterson's mutual exclusion algorithm. In Fig. 1, the algorithm for the case of two processes is modelled in Promela.

```
#define true 1
#define false 0
#define aturn 1
#define bturn 0

bool areq, breq, turn;
bool ain;
bool bin;

proctype a()
{
 do
 :: atomic{ areq=true;
            turn=bturn; }
    (breq==false || turn==aturn);
    ain=true;
    /* critical section cs */
    /* assert(bin==false); */
    ain=false;
    areq=false;
 od
 }

proctype b()
{
 do
 :: atomic{ breq=true;
            turn=aturn; }
    (areq==false || turn==bturn);
    bin=true;
    /* critical section */
    /* assert(ain==false); */
    bin=false;
    breq=false;
 od
}

init
{
   run a(); run b();
}
```

Fig. 1. Peterson's mutual exclusion algorithm

The algorithm uses three global variables `turn`, `areq` and `breq`. The statements on lines 14 and 26, which can only be executed if the boolean equations in them evaluate to true, synchronize the processes. If there is only one process that applies to enter the the critical section, the entry is immediate. Variable `turn` is used to decide which process to enter the critical section when both processes have requested. Variable `ain`(`bin`) is used here to facilitate property specification and it is set `true` iff process `a`(`b`) is in the critical section.

The essential property of a mutual exclusion algorithm is naturally mutual exclusion, which can be easily specified in DQRDC as

$$Req_1 \mathrel{\widehat{=}} \Box(l > 1 \Rightarrow [\![\neg(ain \wedge bin)]\!])$$

The generated corresponding Büchi automaton in Promela is as shown in Fig. 2.

```
 1  never {
 2  T0_init:
 3    if
 4    :: goto T1
 5    fi;
 6  T1:
 7    if
 8    :: ((true))->goto T1
 9    :: ((!ain || !bin))->goto T2
10    :: ((ain && bin))->goto accept_5
11    fi;
12 T2:
13   if
14   :: ((!ain || !bin))->goto T2
15   :: ((ain && bin))->goto accept_5
16   fi;
17 accept_5:
18   if
19   :: ((true))->goto accept_5
20   fi;
21 }
```

Fig. 2. Temporal claim for Req_1.

The mutual exclusion property is simple and can be expressed using assertions (as shown on lines 17 and 29) or a simple monitor process. Next we consider a more involved property,

> *if one process has requested to enter its critical section when the other process is in the critical section, then the first process must have entered the critical section before the second one enters the critical section for another time.*

This property can be expressed as follows w.l.o.g.:

$$Req_2 \mathrel{\widehat{=}} \Box([\![areq \wedge bin]\!]; [\![\neg bin]\!]; [\![bin]\!] \Rightarrow \Diamond[\![ain]\!])$$

The generated Büchi automaton in Promela is shown in Fig. 3.

Taking the temporal claim and the system model as input, the Spin model checker shows that the properties are satisfied.

7 Discussion

In this paper, we have studied the automatic verification of reactive systems against interval logic specifications. Although various interval logics have been

developed for some time, there is little work with similar goal, which is somewhat unusual in view of extensive research regarding LTL. In fact, there have been no clear definitions about what it means for a reactive system to satisfy properties in interval logics. We have considered two classes of common properties and for them defined satisfaction relations that have natural meanings. The two satisfaction relations are unified by a more general one for which model checking support has been provided. We are not clear if the general relation is meaningful for properties outside the two classes and their simple combinations.

Our work is useful to people who prefer to use interval logics. In fact, both LTL and interval logics can be used together in specifying properties of a system. For example, in Peterson's mutual exclusion algorithm case study, one can use the interval logic to specify Req_2 as we have done, since this property may not be easy to express in LTL, but use LTL to specify other properties for which LTL is better or just as good.

Although the algorithm converting the interval logic formulas to finite automata is simple and well-known, there have been few implementations. The early BDD-based implementation by Skakkebæk and Sestoft is integrated into a proof assistant [15] developed on an old version of PVS which has not been supported for years. The more recent implementation by Pandya [12] uses the Mona [6] system by mapping the interval logic to the language of Mona, a monadic second-order logic. Although the implementation of Mona is highly efficient, its sophistication also makes it difficult to handle. What we really need is a package on finite automata, and with Grail+, we have a better control of the implementation. For example, we can control whether to determinize the automaton, which is an expensive operation.

Pandya has also studied model checking reactive systems with interval logics. In his approach, the interval logic does not stand alone and is combined into other logics, for example, into CTL [13]. Pandya has apparently considered combining the interval logic with LTL, and using Spin to check properties expressed in the new logic. The resulting logics are substantially different from existing ones and considerable work may be needed before they are accepted.

```
 1  never {
 2  T0_init:
 3    if
 4    :: goto T0
 5    fi;
 6  T0:
 7    if
 8    :: ((true)) -> goto T0
 9    :: ((areq && bin && !ain))->goto T2
10    fi;
11  T2:
12    if
13    :: ((!bin && !ain)) -> goto T6
14    :: ((areq && bin && !ain))->goto T2
15    fi;
16  T6:
17    if
18    :: ((!bin && !ain)) -> goto T6
19    :: ((bin && !ain)) -> goto accept_14
20    fi;
21  accept_14:
22    if
23    :: ((true)) -> goto accept_19
24    :: ((bin && !ain)) -> goto accept_14
25    fi;
26  accept_19:
27    if
28    :: ((true)) -> goto accept_19
29    fi;
30  }
```

Fig. 3. Temporal claim for Req_2.

References

1. J.R. Buchi. On a Decision Method in Restricted Second Order Arithmetic. In Nagel et al., editor, Logic, Methodology and Philosophy of Science. Stanford Univ. Press, 1960.
2. Bowen Alpern and Fred B. Schneider. Recognizing Safety and Liveness. TR 86–727, 1986.
3. Bowen Alpern and Fred B. Schneider. Defining Liveness. Information Processing Letters, 21:181–185, 1985.
4. Edmund M. Clarke, Orna Grumberg and Doron A. Peled. Model Checking. The MIT Press, 1999.
5. Michael R. Hansen and Zhou Chaochen. Duration Calculus: Logical Foundations. Formal Aspects of Computing, 9:283–330, 1997.
6. J.G. Henriksen, J. Jensen, M. Jorgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic Second-order Logic in practice. In Proc. of TACAS'95, LNCS 1019, Springer-Verlag, 1996.
7. G. Holzmann. The SPIN Model Checker. IEEE Trans. on Software Engineering, 23:279–295, 1997
8. The Grail+ Project. Department of Computer Science, University of Western Ontario, Canada. http://www.csd.uwo.ca/research/grail/
9. Leslie Lamport. Proving the correctness of multiprocess programs. IEEE Transactions on Software Engineering, SE-3(2):125–143, March 1977.
10. B. Moszkowski. A Temporal Logic for Multilevel Reasoning about Hardware. IEEE Computer, 18(2):10–19, 1985.
11. B. Moszkowski. Compositional reasoning about projected and infinite time. Proc. of the First IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'95), IEEE Computer Society Press, p238–245, 1995.
12. P.K. Pandya. Specifying and Deciding Quantified Discrete-Time Duration Calculus Formulae using DCVALID. In Proc. of Workshop on Real-time Tools, Aalborg, Denmark, 2001. TCS00-PKP-1, Tata Institute of Fundamental Research, Mumbai, India, 2000.
13. P.K. Pandya. Model checking CTL[DC]. In Proc. of TACAS 2001, Genova, Italy. LNCS 2031, Springer-Verlag, 2001.
14. Y. S. Ramakrishna, P. M. Melliar-Smith, Louise E. Moser, Laura K. Dillon, G. Kuttys. Interval Logics and Their Decision Procedures, Part I: An Interval Logic. Theoretical Computer Science 166(1&2): 1–47, Elsevier (1996).
15. J.U. Skakkebæk. A Verification Assistant for Real-time Logic. PhD. Thesis, Department of Computer Science, Technical University of Denmark, 1994.
16. Wang Hanpin and Xu Qiwen. Completeness of temporal logics over infinite intervals. Technical Report 158, UNU/IIST, Macau. 1999. Accepted by Applied Discrete Mathematics, Elsevier.
17. P. Wolper. Constructing automata from temporal logic formula: a tutorial. First EEF/Euro Summer School on Trends in Computer Science, LNCS 2090, pp. 261–277, Springer-Verlag, 2001.
18. Pei Yu and Xu Qiwen. Checking Interval Based Properties for Reactive Systems. Technical Report 283, UNU/IIST, Macau. 2003.
19. Zhou Chaochen, Dang Van Hung and Li Xiaoshan. A Duration Calculus with Infinite Intervals, LNCS 965, pp. 16–41, February, 1995.

Widening Operators for Powerset Domains*

Roberto Bagnara[1], Patricia M. Hill[2], and Enea Zaffanella[1]

[1] Department of Mathematics, University of Parma, Italy
{bagnara,zaffanella}@cs.unipr.it
[2] School of Computing, University of Leeds, UK
hill@comp.leeds.ac.uk

Abstract. The *finite powerset construction* upgrades an abstract domain by allowing for the representation of finite disjunctions of its elements. In this paper we define two generic widening operators for the finite powerset abstract domain. Both widenings are obtained by lifting any widening operator defined on the base-level abstract domain and are parametric with respect to the specification of a few additional operators. We illustrate the proposed techniques by instantiating our widenings on powersets of convex polyhedra, a domain for which no non-trivial widening operator was previously known.

1 Introduction

The design and implementation of effective, expressive and efficient abstract domains for data-flow analysis and model-checking is a very difficult task. For this reason, starting with [11], there continues to be strong interest in techniques that derive enhanced abstract domains by applying systematic constructions on simpler, existing domains. Disjunctive completion, direct product, reduced product and reduced power are the first and most famous constructions of this kind [11]; several variations of them as well as others constructions have been proposed in the literature.

Once the carrier of the enhanced abstract domain has been obtained by one of these systematic constructions, the abstract operations can be defined, as usual, as the optimal approximations of the concrete ones. While this completely solves the specification problem, it usually leaves the implementation problem with the designer and gives no guarantees about the efficiency (or even the computability) of the resulting operations. This motivates the importance of generic techniques whereby correct, even though not necessarily optimal, domain operations are derived automatically or semi-automatically from those of the domains the construction operates upon [8,11,18].

This paper focuses on the derivation of widening operators for a kind of disjunctive refinement we call *finite powerset construction*. As far as we know, this

* This work has been partly supported by MURST projects "Aggregate- and Number-Reasoning for Computing: from Decision Algorithms to Constraint Programming with Multisets, Sets, and Maps" and "Constraint Based Verification of Reactive Systems."

B. Steffen and G. Levi (Eds.): VMCAI 2004, LNCS 2937, pp. 135–148, 2004.

is the first time that the problem of deriving non-trivial, provably correct widening operators in a domain refinement is tackled successfully. We also present its specialization to finite powersets of convex polyhedra. Not only is this included to help the reader gain a better intuition regarding the underlying approach but also to provide a definitely non-toy instance that is practically useful for applications such as data-flow analysis and model checking. Sets of polyhedra are implemented in Polylib [24,28] and its successor *PolyLib* [25], even though no widenings are provided. Sets of polyhedra, represented with Presburger formulas made available by the Omega library [23,26], are used in the verifier described in [7]; there, an extrapolation operator (i.e., a widening without convergence guarantee) on sets of polyhedra is described. Another extrapolation operator is implemented in the automated verification tool described in [16], where sets of polyhedra are represented using the clp(q, r) constraint library [22].

The rest of the paper is structured as follows: Section 2 recalls the basic concepts and notations; Section 3 defines the finite powerset construction as a disjunctive refinement for any abstract domain that is a join-semilattice; Section 4 presents two distinct strategies for upgrading any widening for the base-level domain into a widening for the finite powerset domain; Section 5 provides a technique for controlling the precision/efficiency trade-off of these widenings; Section 6 concludes. The proofs of all the stated results can be found in [5].

2 Preliminaries

For a set S, $\wp(S)$ is the powerset of S, whereas $\wp_{\mathrm{f}}(S)$ is the set of all the *finite* subsets of S; the cardinality of S is denoted by $\#\,S$. The first limit ordinal is denoted by ω. Let $\mathcal{O}$ be a set equipped with a well-founded ordering '$\succ$'. If M and N are finite multisets over $\mathcal{O}$, $\#(n, M)$ denotes the number of occurrences of $n \in \mathcal{O}$ in M and $M \gg N$ means that there exists $j \in \mathcal{O}$ such that $\#(j, M) > \#(j, N)$ and, for each $k \in \mathcal{O}$ with $k \succ j$, we have $\#(k, M) = \#(k, N)$. The relation '$\gg$' is well-founded [17].

In this paper we will adopt the abstract interpretation framework proposed in [13, Section 7], where the correspondence between the concrete and the abstract domains is induced from a concrete approximation relation and a concretization function. Since we are not aiming at maximum generality, for the sole purpose of simplifying the presentation, we will consider a particular instance of the framework by assuming a few additional but non-essential domain properties.

The concrete domain is modeled as a complete lattice of semantic properties $\langle C, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$; as usual, the concrete approximation relation $c_1 \sqsubseteq c_2$ holds if c_1 is a stronger property than c_2 (i.e., c_2 approximates c_1). The concrete semantics $c \in C$ of a program is formalized as the least fixpoint of a continuous (concrete) semantic function $\mathcal{F} \colon C \to C$, which is iteratively computed starting from the bottom element, so that $c = \mathcal{F}^{\omega}(\bot)$.

The abstract domain $\hat{D} = \langle D, \vdash, \mathbf{0}, \oplus \rangle$ is modeled as a join-semilattice (i.e., the least upper bound $d_1 \oplus d_2$ exists for all $d_1, d_2 \in D$). We will overload '$\oplus$'

so that, for each $S \in \wp_{\mathrm{f}}(D)$, $\bigoplus S$ denotes the least upper bound of S. The abstract domain $\hat{D}$ is related to the concrete domain by a monotonic and injective concretization function $\gamma\colon D \to C$. Monotonicity and injectivity mean that the abstract partial order '$\vdash$' is indeed the approximation relation induced on D by the concretization function γ. For all $d_1, d_2 \in D$, we will use the notation $d_1 \Vdash d_2$ to mean that $d_1 \vdash d_2$ and $d_1 \neq d_2$. We assume the existence of a monotonic abstract semantic function $\mathcal{F}^\sharp\colon D \to D$ that is sound with respect to $\mathcal{F}\colon C \to C$:

$$\forall c \in C : \forall d \in D : c \sqsubseteq \gamma(d) \implies \mathcal{F}(c) \sqsubseteq \gamma\big(\mathcal{F}^\sharp(d)\big). \tag{1}$$

This local correctness condition ensures that each concrete iterate can be safely approximated by computing the corresponding abstract iterate (starting from the bottom element $\mathbf{0} \in D$). However, due to the weaker algebraic properties satisfied by the abstract domain, the abstract upward iteration sequence may not converge. Even when it converges, it may fail to do so in a finite number of steps, therefore being useless for the purposes of static analysis.

Widening operators [9,10,13,14] provide a simple and general characterization for enforcing and accelerating convergence. We will adopt a minor variation of the classical definition of widening operator (see footnote 6 in [14, p. 275]).

Definition 1. (Widening.) *Let $\langle D, \vdash, \mathbf{0}, \oplus\rangle$ be a join-semilattice. The partial operator $\nabla\colon D \times D \rightarrowtail D$ is a* widening *if*

1. *$d_1 \vdash d_2$ implies $d_2 \vdash d_1 \nabla d_2$ for each $d_1, d_2 \in D$;*
2. *for each increasing chain $d_0 \vdash d_1 \vdash \cdots$, the increasing chain defined by $d'_0 := d_0$ and $d'_{i+1} := d'_i \nabla (d'_i \oplus d_{i+1})$, for $i \in \mathbb{N}$, is not strictly increasing.*

Any widening operator '∇' induces a corresponding partial ordering '$\vdash_\nabla$' on the domain D; this is defined as the reflexive and transitive closure of the relation $\big\{\, (d_1, d) \in D \times D \;\big|\; \exists d_2 \in D \,.\, d_1 \Vdash d_2 \wedge d = d_1 \nabla d_2 \,\big\}$. The relation '$\vdash_\nabla$' satisfies the ascending chain condition. We write $d_1 \Vdash_\nabla d_2$ to denote $d_1 \vdash_\nabla d_2 \wedge d_1 \neq d_2$.

It can be proved that the *upward iteration sequence with widenings* starting at the bottom element $d_0 := \mathbf{0}$ and defined by

$$d_{i+1} := \begin{cases} d_i, & \text{if } \mathcal{F}^\sharp(d_i) \vdash d_i, \\ d_i \nabla \big(d_i \oplus \mathcal{F}^\sharp(d_i)\big), & \text{otherwise,} \end{cases}$$

converges after a finite number $j \in \mathbb{N}$ of iterations [14]. Note that the widening is always applied to arguments $d = d_i$ and $d' = d_i \oplus \mathcal{F}^\sharp(d_i)$ satisfying $d \Vdash d'$. Also, when condition (1) holds, the post-fixpoint $d_j \in D$ of $\mathcal{F}^\sharp$ is a correct approximation of the concrete semantics, i.e., $\mathcal{F}^\omega(\bot) \sqsubseteq \gamma(d_j)$.

2.1 The Abstract Domain of Polyhedra

We now instantiate the abstract interpretation framework sketched above by presenting the well-known abstract domain of closed convex polyhedra. This

domain will be used throughout the paper to illustrate the generic widening techniques that will be defined.

Let $\mathbb{R}^n$, where $n > 0$, be the n-dimensional real vector space. The set $\mathcal{P} \subseteq \mathbb{R}^n$ is a *closed and convex polyhedron* (*polyhedron*, for short) if and only if $\mathcal{P}$ can be expressed as the intersection of a finite number of closed affine half-spaces of $\mathbb{R}^n$. The set $\mathbb{CP}_n$ of closed convex polyhedra on $\mathbb{R}^n$, when partially ordered by subset inclusion, is a lattice having the empty set and $\mathbb{R}^n$ as the bottom and top elements, respectively; the binary meet operation is set-intersection, whereas the binary join operation, denoted by '$\uplus$', is called *convex polyhedral hull* (*poly-hull*, for short). Therefore, we have the abstract domain

$$\widehat{\mathbb{CP}}_n := \langle \mathbb{CP}_n, \subseteq, \varnothing, \mathbb{R}^n, \uplus, \cap \rangle.$$

This domain can be related to several concrete domains, depending on the intended application. One example of a concrete domain is the complete lattice

$$\hat{\mathsf{A}}_n := \big\langle \wp(\mathbb{R}^n), \subseteq, \varnothing, \mathbb{R}^n, \cup, \cap \big\rangle.$$

Note that $\widehat{\mathbb{CP}}_n$ is a meet-sublattice of $\hat{\mathsf{A}}_n$, sharing the same bottom and top elements. Another example is the complete lattice $\hat{\mathsf{B}}_n := \big\langle \wp_c(\mathbb{R}^n), \subseteq, \varnothing, \mathbb{R}^n, \cup_c, \cap \big\rangle$, where $\wp_c(\mathbb{R}^n)$ is the set of all topologically closed and convex subsets of $\mathbb{R}^n$ and the join operation '$\cup_c$' returns the smallest topologically closed and convex set containing its arguments. As a final example of concrete domain for some analysis, consider the complete lattice $\hat{\mathsf{C}}_n := \big\langle \wp(\mathbb{CP}_n), \subseteq, \varnothing, \mathbb{CP}_n, \cup, \cap \big\rangle$.

The abstract domain $\widehat{\mathbb{CP}}_n$, which is a join-semilattice, is related to the concrete domains shown above by the concretization functions $\gamma^{\mathsf{A}} \colon \mathbb{CP}_n \to \wp(\mathbb{R}^n)$, $\gamma^{\mathsf{B}} \colon \mathbb{CP}_n \to \wp_c(\mathbb{R}^n)$ and $\gamma^{\mathsf{C}} \colon \mathbb{CP}_n \to \wp(\mathbb{CP}_n)$: for each $\mathcal{P} \in \mathbb{CP}_n$, we have both $\gamma^{\mathsf{A}}(\mathcal{P}) := \mathcal{P}$ and $\gamma^{\mathsf{B}}(\mathcal{P}) := \mathcal{P}$, and $\gamma^{\mathsf{c}}(\mathcal{P}) := \downarrow\mathcal{P} := \{\, \mathcal{Q} \in \mathbb{CP}_n \mid \mathcal{Q} \subseteq \mathcal{P} \,\}$. All these concretization functions are trivially monotonic and injective.

For each choice of the concrete domain $C \in \big\{ \wp(\mathbb{R}^n), \wp_c(\mathbb{R}^n), \wp(\mathbb{CP}_n) \big\}$, the continuous semantic function $\mathcal{F} \colon C \to C$ and the corresponding monotonic abstract semantic function $\mathcal{F}^\sharp \colon \mathbb{CP}_n \to \mathbb{CP}_n$, which is assumed to be correct, are deliberately left unspecified. The domain $\widehat{\mathbb{CP}}_n$ contains infinite ascending chains having no least upper bound in $\mathbb{CP}_n$. Thus, the convergence of the abstract iteration sequence has to be guaranteed by the adoption of widening operators.

The first widening on polyhedra was introduced in [15] and refined in [19]. This operator, denoted by '∇_s', has been termed *standard widening* and used almost universally. In [4], we presented a framework designed so that all its instances are widening operators on $\mathbb{CP}_n$. The standard widening '∇_s' is an instance of the framework and all the other instances, including the specific widening '$\hat{\nabla}$' defined and experimentally evaluated in [4], are at least as precise as '∇_s'. For a formal definition of both '∇_s' and '$\hat{\nabla}$', we refer the reader to [4].

3 A Disjunctive Refinement

In this section, we present the *finite powerset* operator, which is a domain refinement similar to disjunctive completion [11] and is obtained by a variant of

the *down-set completion* construction presented in [12]. The following notation and definitions are mainly borrowed from [2, Section 6].

Definition 2. (Non-redundancy.) *Let* $\hat{D} = \langle D, \vdash, \mathbf{0}, \oplus \rangle$ *be a join-semilattice. The set* $S \in \wp(D)$ *is called* non-redundant *with respect to '*$\vdash$*' if and only if* $\mathbf{0} \notin S$ *and* $\forall d_1, d_2 \in S : d_1 \vdash d_2 \implies d_1 = d_2$. *The set of finite non-redundant subsets of* D *(with respect to '*$\vdash$*') is denoted by* $\wp_{\mathrm{fn}}(D, \vdash)$. *The reduction function* $\Omega^{\vdash}_{D} \colon \wp_{\mathrm{f}}(D) \to \wp_{\mathrm{fn}}(D, \vdash)$ *mapping each finite set into its non-redundant counterpart is defined, for each* $S \in \wp_{\mathrm{f}}(D)$, *by*

$$\Omega^{\vdash}_{D}(S) := S \setminus \{\, d \in S \mid d = \mathbf{0} \vee \exists d' \in S \,.\, d \Vdash d' \,\}.$$

The restriction to the finite subsets reflects the fact that here we are mainly interested in an abstract domain where disjunctions are implemented by explicit collections of elements of the base-level abstract domain.

Definition 3. (Finite powerset domain.) *Let* $\hat{D} := \langle D, \vdash, \mathbf{0}, \oplus \rangle$ *be a join-semilattice. The* finite powerset domain *over* $\hat{D}$ *is the join-semilattice*

$$\hat{D}_{\mathrm{P}} := \big\langle \wp_{\mathrm{fn}}(D, \vdash), \vdash_{\mathrm{P}}, \mathbf{0}_{\mathrm{P}}, \oplus_{\mathrm{P}} \big\rangle,$$

where $\mathbf{0}_{\mathrm{P}} := \varnothing$ *and, for all* $S_1, S_2 \in \wp_{\mathrm{fn}}(D, \vdash)$, $S_1 \oplus_{\mathrm{P}} S_2 := \Omega^{\vdash}_{D}(S_1 \cup S_2)$.

The approximation ordering '$\vdash_{\mathrm{P}}$' induced by '$\oplus_{\mathrm{P}}$' is the Hoare powerdomain partial order [1], so that $S_1 \vdash_{\mathrm{P}} S_2$ if and only if $\forall d_1 \in S_1 : \exists d_2 \in S_2 \,.\, d_1 \vdash d_2$. A sort of Egli-Milner partial order relation [1] will also be useful: $S_1 \vdash_{\mathrm{EM}} S_2$ holds if and only if either $S_1 = \mathbf{0}_{\mathrm{P}}$ or $S_1 \vdash_{\mathrm{P}} S_2$ and $\forall d_2 \in S_2 : \exists d_1 \in S_1 \,.\, d_1 \vdash d_2$. An *(Egli-Milner) connector* for $\hat{D}_{\mathrm{P}}$, denoted by '$\boxplus_{\mathrm{EM}}$' is any upper bound operator for the Egli-Milner ordering on $\wp_{\mathrm{fn}}(D, \vdash)$. Note that although a *least* upper bound for '$\vdash_{\mathrm{EM}}$' may not exist, a connector can always be defined; for instance, we can let $S_1 \boxplus_{\mathrm{EM}} S_2 := \big\{ \bigoplus (S_1 \cup S_2) \big\}$.

Besides the requirement on finiteness, another difference with respect to the down-set completion of [12] is that we are dropping the assumption about the complete distributivity of the concrete domain. This is possible because our semantic domains are not necessarily related by Galois connections, so that this property does not have to be preserved.

The finite powerset domain is related to the concrete domain by means of the concretization function $\gamma_{\mathrm{P}} \colon \wp_{\mathrm{fn}}(D, \vdash) \to C$ defined by

$$\gamma_{\mathrm{P}}(S) := \bigsqcup \big\{\, \gamma(d) \;\big|\; d \in S \,\big\}.$$

Note that γ_{P} is monotonic but not necessarily injective. For $S_1, S_2 \in \wp_{\mathrm{fn}}(D, \vdash)$, we write $S_1 \equiv_{\gamma_{\mathrm{P}}} S_2$ to denote that the two abstract elements actually denote the same concrete element, i.e., when $\gamma_{\mathrm{P}}(S_1) = \gamma_{\mathrm{P}}(S_2)$. It is easy to see that '$\equiv_{\gamma_{\mathrm{P}}}$' is a congruence relation on $\hat{D}_{\mathrm{P}}$. As noted in [12], non-redundancy only provides a partial, syntactic form of reduction. On the other hand, requiring the full, semantic form of reduction for a finite powerset domain can be computationally very expensive.

A correct abstract semantic function $\mathcal{F}^\sharp_{\mathrm{P}}\colon \wp_{\mathrm{fn}}(D,\vdash) \to \wp_{\mathrm{fn}}(D,\vdash)$ on the finite powerset domain may be provided by an ad-hoc definition. More often, if the concrete semantic function $\mathcal{F}\colon C \to C$ satisfies suitable hypotheses, $\mathcal{F}^\sharp_{\mathrm{P}}$ can be safely induced from the abstract semantic function $\mathcal{F}^\sharp\colon D \to D$. For instance, if $\mathcal{F}$ is additive, we can define $\mathcal{F}^\sharp_{\mathrm{P}}$ as follows [11,18]:

$$\mathcal{F}^\sharp_{\mathrm{P}}(S) := \Omega^{\vdash}_{D}\Big(\big\{\, \mathcal{F}^\sharp(d) \;\big|\; d \in S \,\big\}\Big).$$

3.1 The Finite Powerset Domain of Polyhedra

The polyhedral domain $(\widehat{\mathbb{CP}}_n)_{\mathrm{P}}$, having carrier $\wp_{\mathrm{fn}}(\mathbb{CP}_n, \subseteq)$, is the finite powerset domain over $\widehat{\mathbb{CP}}_n$. The approximation ordering is '$\subseteq_{\mathrm{P}}$' where, for each $\mathcal{S}_1, \mathcal{S}_2 \in \wp_{\mathrm{fn}}(\mathbb{CP}_n, \subseteq)$,

$$\mathcal{S}_1 \subseteq_{\mathrm{P}} \mathcal{S}_2 \iff \forall \mathcal{P}_1 \in \mathcal{S}_1 : \exists \mathcal{P}_2 \in \mathcal{S}_2 \,.\, \mathcal{P}_1 \subseteq \mathcal{P}_2.$$

Let $\gamma^{\mathsf{A}}_{\mathrm{P}}$, $\gamma^{\mathsf{B}}_{\mathrm{P}}$ and $\gamma^{\mathsf{C}}_{\mathrm{P}}$ denote the (powerset) concretization functions induced by γ^{A}, γ^{B} and γ^{C}, respectively. Then, the relation '$\equiv_{\gamma^{\mathsf{A}}_{\mathrm{P}}}$' makes two finite sets of polyhedra equivalent if and only if they have the same set-union. The general problem of deciding the semantic equivalence with respect to $\gamma^{\mathsf{A}}_{\mathrm{P}}$ of two finite (non-redundant) collections of polyhedra is known to be computationally hard [27]. For $\gamma^{\mathsf{B}}_{\mathrm{P}}$, the relation '$\equiv_{\gamma^{\mathsf{B}}_{\mathrm{P}}}$' makes two finite sets of polyhedra equivalent if and only if they have the same poly-hull, so that the powerset construction provides no benefit at all. Finally, $\gamma^{\mathsf{C}}_{\mathrm{P}}$ is injective so that '$\equiv_{\gamma^{\mathsf{C}}_{\mathrm{P}}}$' coincides with the identity congruence relation.

Example 1. For the polyhedral domain $(\widehat{\mathbb{CP}}_1)_{\mathrm{P}}$, let[1]

$$\begin{aligned}
\mathcal{T}_0 &:= \big\{\{0 \le x \le 2\}, \{1 \le x \le 2\}, \{3 \le x \le 4\}, \{4 \le x \le 5\}\big\},\\
\mathcal{T}_1 &:= \big\{\{0 \le x \le 2\}, \{3 \le x \le 4\}, \{4 \le x \le 5\}\big\},\\
\mathcal{T}_2 &:= \big\{\{0 \le x \le 1\}, \{1 \le x \le 2\}, \{3 \le x \le 5\}\big\}.
\end{aligned}$$

Then $\mathcal{T}_0 \notin \wp_{\mathrm{fn}}(\mathbb{CP}_1, \subseteq)$, but $\mathcal{T}_1 = \Omega^{\subseteq}_{\mathbb{CP}_1}(\mathcal{T}_0) \in \wp_{\mathrm{fn}}(\mathbb{CP}_1, \subseteq)$. Also, $\mathcal{T}_1 \equiv_{\gamma^{\mathsf{A}}_{\mathrm{P}}} \mathcal{T}_2$.

4 Widening the Finite Powerset Domain

If the domain refinement of the previous section is meant to be used for static analysis, then a key ingredient that is still missing is a systematic way of ensuring the termination of the analysis. In this section, we describe two widening strategies that rely on the existence of a widening $\nabla\colon D \times D \rightarrowtail D$ on the base-level abstract domain.[2] We start by proposing a general specification of an extrapolation operator that lifts this '∇' operator to the powerset domain.

[1] In this and the following examples, a polyhedron $\mathcal{P} \in \mathbb{CP}_n$ will be denoted by a corresponding finite set of linear equality and non-strict inequality constraints.

[2] If the base-level abstract domain $\hat{D}$ is finite or Noetherian, so that it is not necessarily endowed with an explicit widening operator, then a dummy widening can be obtained by considering the least upper bound operator '$\oplus$'.

Definition 4. (The ∇-connected extrapolation heuristics.) *A partial operator $h^{\nabla}_{\mathrm{P}}\colon \wp_{\mathrm{fn}}(D,\vdash)^2 \rightarrowtail \wp_{\mathrm{fn}}(D,\vdash)$ is a* ∇-connected extrapolation heuristics for $\hat{D}_{\mathrm{P}}$ *if, for all $S_1, S_2 \in \wp_{\mathrm{fn}}(D,\vdash)$ such that $S_1 \Vdash_{\mathrm{P}} S_2$, $h^{\nabla}_{\mathrm{P}}(S_1,S_2)$ is defined and satisfies the following conditions:*

$$S_2 \vdash_{\mathrm{EM}} h^{\nabla}_{\mathrm{P}}(S_1,S_2); \tag{2}$$

$$\forall d \in h^{\nabla}_{\mathrm{P}}(S_1,S_2) \setminus S_2 : \exists d_1 \in S_1 \,.\, d_1 \Vdash_{\nabla} d; \tag{3}$$

$$\forall d \in h^{\nabla}_{\mathrm{P}}(S_1,S_2) \cap S_2 : \big((\exists d_1 \in S_1 \,.\, d_1 \Vdash d) \rightarrow (\exists d'_1 \in S_1 \,.\, d'_1 \Vdash_{\nabla} d)\big). \tag{4}$$

Informally, condition (2) ensures that the result is an upper approximation of S_2 in which every element covers at least one element of S_2 (i.e., the heuristics cannot add elements that are unrelated to S_2); conditions (3) and (4) ensure that in the resulting set, each element covering an element of S_1 originates from an application of '∇' to a (possibly different) element of S_1.

It is straightforward to construct an algorithm for computing a ∇-connected extrapolation heuristics for any given base-level widening '∇'.

Proposition 1. *For all $S_1, S_2 \in \wp_{\mathrm{fn}}(D,\vdash)$ such that $S_1 \vdash_{\mathrm{P}} S_2$, let*

$$h^{\nabla}_{\mathrm{P}}(S_1,S_2) := S_2 \oplus_{\mathrm{P}} \Omega^{\vdash}_{D}\big(\{\, d_1 \nabla d_2 \in D \mid d_1 \in S_1, d_2 \in S_2, d_1 \Vdash d_2 \,\}\big).$$

Then 'h^{∇}_{P}' is a ∇-connected extrapolation heuristics for $\hat{D}_{\mathrm{P}}$.

For the finite powerset domain over $\widehat{\mathbb{CP}}_n$, lines 10–15 of the algorithm specified in [7, Figure 8, page 773] provide an implementation of the heuristics 'h^{∇}_{P}' defined in Proposition 1, instantiated with the standard widening, '∇_s', on $\widehat{\mathbb{CP}}_n$.

Example 2. To see that the 'h^{∇}_{P}' defined in Proposition 1 is not a widening for $(\widehat{\mathbb{CP}}_n)_{\mathrm{P}}$, consider the strictly increasing sequence $\mathcal{T}_0 \subsetneq_{\mathrm{P}} \mathcal{T}_1 \subsetneq_{\mathrm{P}} \cdots$ in $\mathbb{CP}_1$ defined by $\mathcal{T}_j := \{\, \mathcal{P}_i \mid 0 \leq i \leq j \,\}$, where $\mathcal{P}_i := \{x = i\}$, for $i \in \mathbb{N}$. Then, no matter what the specification for '∇' is, we obtain $h^{\nabla}_{\mathrm{P}}(\mathcal{T}_j, \mathcal{T}_{j+1}) = \mathcal{T}_{j+1}$, for all $j \in \mathbb{N}$. Thus, the "widened" sequence is diverging.

4.1 Powerset Widenings Using Egli-Milner Connectors

Example 2 shows that, when computing $h^{\nabla}_{\mathrm{P}}(S_1,S_2)$, divergence is caused by those elements of S_2 that cover none of the elements occurring in S_1, i.e., when $S_1 \nvdash_{\mathrm{EM}} S_2$. Thus, stabilization can be obtained by replacing S_2 with $S_1 \boxplus_{\mathrm{EM}} S_2$, where '$\boxplus_{\mathrm{EM}}$' is a connector for $\hat{D}_{\mathrm{P}}$. We therefore define a simple widening operator on the finite powerset domain that uses a connector to ensure termination.

Definition 5. (The '$_{\mathrm{EM}}\nabla_{\mathrm{P}}$' widening.) *Let '$h^{\nabla}_{\mathrm{P}}$' be a ∇-connected extrapolation heuristics and '$\boxplus_{\mathrm{EM}}$' be a connector for $\hat{D}_{\mathrm{P}}$. Let also $S_1, S_2 \in \wp_{\mathrm{fn}}(D,\vdash)$, where $S_1 \Vdash_{\mathrm{P}} S_2$. Then*

$$S_1 \,{}_{\mathrm{EM}}\nabla_{\mathrm{P}}\, S_2 := h^{\nabla}_{\mathrm{P}}(S_1, S'_2), \quad \textit{where } S'_2 := \begin{cases} S_2, & \textit{if } S_1 \vdash_{\mathrm{EM}} S_2; \\ S_1 \boxplus_{\mathrm{EM}} S_2, & \textit{otherwise.} \end{cases}$$

Theorem 1. *The '$_{\text{EM}}\nabla_{\text{P}}$' operator is a widening on $\hat{D}_{\text{P}}$.*

Example 3. To illustrate the widening operator '$_{\text{EM}}\nabla_{\text{P}}$' we consider the powerset domain $(\widehat{\mathbb{CP}}_1)_{\text{P}}$, with the standard widening '∇_s' on $\widehat{\mathbb{CP}}_1$ and the trivial connector '$\uplus_{\text{EM}}$' returning the singleton poly-hull of its arguments. Consider the sequence $\mathcal{T}_0 \subseteq_{\text{P}} \mathcal{T}_1 \subseteq_{\text{P}} \cdots$ of Example 2 and the widened sequence $\mathcal{U}_0 \subseteq_{\text{P}} \mathcal{U}_1 \subseteq_{\text{P}} \cdots$ where $\mathcal{U}_0 = \mathcal{T}_0$ and $\mathcal{U}_i = \mathcal{U}_{i-1}\, {}_{\text{EM}}\nabla_{\text{P}}(\mathcal{U}_{i-1} \uplus_{\text{P}} \mathcal{T}_i)$, for each $i > 0$. When computing $\mathcal{U}_1$, the second argument of the widening is $\mathcal{U}_0 \uplus_{\text{P}} \mathcal{T}_1 = \mathcal{T}_1$. Note that $\mathcal{U}_0 \vdash_{\text{EM}} \mathcal{T}_1$ does not hold so that the connector is needed. Thus, we obtain

$$\mathcal{U}_1 = h^{\nabla}_{\text{P}}(\mathcal{U}_0, \mathcal{U}_0 \uplus_{\text{EM}} \mathcal{T}_1) = h^{\nabla}_{\text{P}}\Big(\mathcal{U}_0, \big\{\{0 \le x \le 1\}\big\}\Big) = \big\{\{0 \le x\}\big\}.$$

In the next iteration we obtain stabilization. Clearly, in general the precision of this widening will depend on the chosen connector operator.

For the polyhedral domain $\widehat{\mathbb{CP}}_n$, the powerset widening '$_{\text{EM}}\nabla_{\text{P}}$' using the '$h^{\nabla}_{\text{P}}$' heuristics defined in Proposition 1 is similar to but not quite the same as the operator sketched in [7]. As noted in that paper, the algorithm in [7, Figure 8, page 773] cannot ensure the termination of the analysis. To this end, instead of using a connector operator it is proposed that, when the cardinality of the abstract collection reaches a fixed threshold, a further poly-hull approximation be applied. However, there are examples indicating that such an approach cannot come with a termination guarantee when considering arbitrary increasing sequences [5].

4.2 Powerset Widenings Using Finite Convergence Certificates

We now present another widening operator (denoted here by '$_{\mu}\nabla_{\text{P}}$') for the finite powerset domain. This requires that the operator '∇' defined on the base-level domain is provided with a (finitely computable) finite convergence certificate. Formally, a *finite convergence certificate* for '∇' (on $\hat{D}$) is a triple $(\mathcal{O}, \succ, \mu)$ where $(\mathcal{O}, \succ)$ is a well-founded ordered set and $\mu\colon D \to \mathcal{O}$, which is called *level mapping*, is such that, for all $d_1 \Vdash d_2 \in D$, $\mu(d_1) \succ \mu(d_1 \nabla d_2)$. We will abuse notation by writing μ to denote the certificate $(\mathcal{O}, \succ, \mu)$.

Example 4. For the polyhedral domain $\widehat{\mathbb{CP}}_n$ and the standard widening '∇_s', we can define a certificate $(\mathcal{O}_s, \succ_s, \mu_s)$ where $\mathcal{O}_s$ is the pair $(\mathbb{N}, \mathbb{N})$, '$\succ_s$' the lexicographic ordering of the pair using $>$ for the individual ordering of the components and $\mu_s\colon \mathbb{CP}_n \to \mathcal{O}_s$ the level mapping $\mu_s(\mathcal{P}) = \big(n - \dim(\mathcal{P}), k\big)$, where $\dim(\mathcal{P})$ is the dimension of $\mathcal{P}$ and k the minimal number of half-spaces needed to define $\mathcal{P}$. Similarly, a certificate for the widening '$\hat{\nabla}$' on $\widehat{\mathbb{CP}}_n$ proposed in [4] can be obtained by considering the level mapping $\mu_b\colon \mathbb{CP}_n \to \mathcal{O}_b$ induced by the *limited growth ordering* relation '$\curvearrowright$' defined in [4], so that we have $\mu_b(\mathcal{P}_1) \succ_b \mu_b(\mathcal{P}_2)$ if and only if $\mathcal{P}_1 \curvearrowright \mathcal{P}_2$. For lack of space, we refer the reader to [4].

Given a certificate for '∇', we can define a suitable limited growth ordering relation '$\frown_{\mathrm{P}}$' for the finite powerset domain $\hat{D}_{\mathrm{P}}$ that satisfies the ascending chain condition.

Definition 6. (The '$\frown_{\mathrm{P}}$' relation.) *The relation $\frown_{\mathrm{P}} \subseteq \wp_{\mathrm{fn}}(D,\vdash) \times \wp_{\mathrm{fn}}(D,\vdash)$ induced by the certificate μ for '∇' is such that, for each $S_1, S_2 \in \wp_{\mathrm{fn}}(D,\vdash)$, $S_1 \frown_{\mathrm{P}} S_2$ if and only if either one of the following conditions holds:*

$$\mu(\bigoplus S_1) \succ \mu(\bigoplus S_2); \tag{5}$$

$$\mu(\bigoplus S_1) = \mu(\bigoplus S_2) \wedge \# S_1 > 1 \wedge \# S_2 = 1; \tag{6}$$

$$\mu(\bigoplus S_1) = \mu(\bigoplus S_2) \wedge \# S_1 > 1 \wedge \# S_2 > 1 \wedge \tilde{\mu}(S_1) \gg \tilde{\mu}(S_2) \tag{7}$$

where, for each $S \in \wp_{\mathrm{fn}}(D,\vdash)$, $\tilde{\mu}(S)$ denotes the multiset over $\mathcal{O}$ obtained by applying μ to each abstract element in S.

Proposition 2. *The '$\frown_{\mathrm{P}}$' relation satisfies the ascending chain condition.*

Intuitively, the relation '$\frown_{\mathrm{P}}$' will induce a certificate $\mu_{\mathrm{P}} \colon \hat{D}_{\mathrm{P}} \to \mathcal{O}_{\mathrm{P}}$ for the new widening. Namely, by defining $\mu_{\mathrm{P}}(S_1) \succ_{\mathrm{P}} \mu_{\mathrm{P}}(S_2)$ if and only if $S_1 \frown_{\mathrm{P}} S_2$, we will obtain $\mu_{\mathrm{P}}(S_1) \succ_{\mathrm{P}} \mu_{\mathrm{P}}(S_1 \,{}_{\mu}\nabla_{\mathrm{P}}\, S_2)$.

The specification of our "certificate-based widening" assumes the existence of a *subtract* operation for the base-level domain. It is expected that a specific subtraction would be provided for each domain; here we just indicate a minimal specification.

Definition 7. (Subtraction.) *The partial operator $\ominus \colon D \times D \rightarrowtail D$ is a* subtraction *for $\hat{D}$ if, for all $d_1, d_2 \in D$ such that $d_2 \vdash d_1$, we have $d_1 \ominus d_2 \vdash d_1$ and $d_1 = (d_1 \ominus d_2) \oplus d_2$.*

A trivial subtraction operator can always be defined as $d_1 \ominus d_2 := d_1$.

Example 5. In $\widehat{\mathbb{CP}}_n$, the function $\mathrm{diff} \colon \mathbb{CP}_n \times \mathbb{CP}_n \to \mathbb{CP}_n$ is defined so that, for any $\mathcal{P}, \mathcal{Q} \in \mathbb{CP}_n$, $\mathrm{diff}(\mathcal{P}, \mathcal{Q})$ denotes the smallest closed and convex polyhedron containing the set difference $\mathcal{P} \setminus \mathcal{Q}$. Then, if $\mathcal{Q} \subseteq \mathcal{P}$, we have $\mathrm{diff}(\mathcal{P}, \mathcal{Q}) \subseteq \mathcal{P}$ and

$$\mathcal{P} = (\mathcal{P} \setminus \mathcal{Q}) \cup \mathcal{Q} = \mathrm{diff}(\mathcal{P}, \mathcal{Q}) \cup \mathcal{Q} = \mathrm{diff}(\mathcal{P}, \mathcal{Q}) \uplus \mathcal{Q},$$

so that 'diff' is a subtraction.

We can now define the *certificate-based widening* '$_{\mu}\nabla_{\mathrm{P}}$'.

Definition 8. (The '$_{\mu}\nabla_{\mathrm{P}}$' widening.) *Let '$\frown_{\mathrm{P}}$' be the limited growth ordering induced by the certificate μ for '∇' and let '$\boxplus_{\mathrm{P}}$' be any upper bound operator on $\hat{D}_{\mathrm{P}}$. Let $S_1, S_2 \in \wp_{\mathrm{fn}}(D,\vdash)$ be such that $S_1 \Vdash_{\mathrm{P}} S_2$. Also, if $\bigoplus S_1 \Vdash \bigoplus(S_1 \boxplus_{\mathrm{P}} S_2)$, let $d \in D$ be defined as $d := \big(\bigoplus S_1 \nabla \bigoplus(S_1 \boxplus_{\mathrm{P}} S_2)\big) \ominus \big(\bigoplus(S_1 \boxplus_{\mathrm{P}} S_2)\big)$. Then*

$$S_1 \,{}_{\mu}\nabla_{\mathrm{P}}\, S_2 := \begin{cases} S_1 \boxplus_{\mathrm{P}} S_2, & \text{if } S_1 \frown_{\mathrm{P}} S_1 \boxplus_{\mathrm{P}} S_2; \\ (S_1 \boxplus_{\mathrm{P}} S_2) \oplus_{\mathrm{P}} \{d\}, & \text{if } \bigoplus S_1 \Vdash \bigoplus(S_1 \boxplus_{\mathrm{P}} S_2); \\ \{\bigoplus S_2\}, & \text{otherwise.} \end{cases}$$

In the first case, we simply return the upper bound $S_1 \boxplus_{\mathrm{P}} S_2$, since this is enough to ensure a strict decrease in the level mapping. In the second case, the join of S_1 is strictly more precise than the join of $S_1 \boxplus_{\mathrm{P}} S_2$, so that we apply '$\nabla$' to them and then, using the subtraction operator, improve the obtained result, since $S_1 \rightharpoonup_{\mathrm{P}} (S_1 \boxplus_{\mathrm{P}} S_2) \oplus_{\mathrm{P}} \{d\}$ holds. In the last case, since the join of $S_1 \boxplus_{\mathrm{P}} S_2$ is invariant, we return the singleton consisting of the join itself, as originally proposed in [11, Section 9].

Theorem 2. *The '$_{\mu}\nabla_{\mathrm{P}}$' operator is a widening on $\hat{D}_{\mathrm{P}}$.*

Example 6. To illustrate the last two cases of Definition 8, consider the domain $(\widehat{\mathbb{CP}}_1)_{\mathrm{P}}$ with the standard widening '∇_s' for $\widehat{\mathbb{CP}}_1$ certified by the level mapping μ_s defined in Example 4 and the upper bound operator '$\boxplus_{\mathrm{P}}$' defined as '$\oplus_{\mathrm{P}}$' so that $S_1 \boxplus_{\mathrm{P}} S_2 = S_2$ always holds.

Let $\mathcal{T}_1 = \{\{0 \le x \le 1\}\}$ and $\mathcal{T}_2 = \{\{0 \le x \le 1\}, \{2 \le x \le 3\}\}$. Then $\mathcal{T}_1 \not\rightharpoonup_{\mathrm{P}} \mathcal{T}_2$, so that the condition for the first case in Definition 8 does not hold. The poly-hulls of $\mathcal{T}_1$ and $\mathcal{T}_2$ are $\{0 \le x \le 1\}$ and $\{0 \le x \le 3\}$, respectively, so that the condition for the second case holds. Since $\biguplus \mathcal{T}_1 \nabla_s \biguplus \mathcal{T}_2 = \{0 \le x\}$, then by letting the polyhedron $\mathcal{P}$ be the element d as specified in Definition 8, we obtain $\mathcal{P} = \mathrm{diff}\big(\{0 \le x\}, \{0 \le x \le 3\}\big) = \{3 \le x\}$, so that

$$\mathcal{T}_1 \,{}_{\mu}\nabla_{\mathrm{P}} \mathcal{T}_2 = \mathcal{T}_2 \uplus_{\mathrm{P}} \{\mathcal{P}\} = \big\{\{0 \le x \le 1\}, \{2 \le x \le 3\}, \{3 \le x\}\big\}.$$

Now let $\mathcal{T}_3 = \big\{\{x = 1\}, \{x = 3\}\big\}$ and $\mathcal{T}_4 = \big\{\{x = 1\}, \{x = 2\}, \{x = 3\}\big\}$. Then $\mathcal{T}_3 \not\rightharpoonup_{\mathrm{P}} \mathcal{T}_4$, so that the condition for the first case in Definition 8 does not hold. Moreover, $\biguplus \mathcal{T}_3 = \biguplus \mathcal{T}_4 = \{1 \le x \le 3\}$, so that neither the second case applies. Thus, $\mathcal{T}_3 \,{}_{\mu}\nabla_{\mathrm{P}} \mathcal{T}_4 = \big\{\{1 \le x \le 3\}\big\}$.

As shown in the example above, Definition 8 does not require that the upper bound operator '$\boxplus_{\mathrm{P}}$' is based on the base-level widening '∇'. Moreover, the scheme of Definition 8 can be easily extended to any finite set of heuristically chosen upper bound operators on $\hat{D}_{\mathrm{P}}$, still obtaining a proper widening operator. The simplest heuristics, already used in the example above, is the one taking $\boxplus_{\mathrm{P}} := \oplus_{\mathrm{P}}$. If this fails to ensure a decrease in the level mapping, another possibility is the adoption of a ∇-connected extrapolation heuristics 'h^{∇}_{P}' for $\hat{D}_{\mathrm{P}}$. Anyway, many variations could be defined, depending on the required precision/efficiency trade-off. In the following section, we investigate one of these possibilities, which originates as a generalization of an idea proposed in [7].

5 Merging Elements According to a Congruence Relation

When computing a powerset widening $S_1 \nabla_{\mathrm{P}} S_2$, no matter if it is based on an Egli-Milner connector or a finite convergence certificate, some of the elements occurring in the second argument S_2 can be *merged together* (i.e., joined) without affecting the finite convergence guarantee. This merging operation can be guided

by a congruence relation on the finite powerset domain $\hat{D}_{\mathrm{P}}$, the idea being that a well-chosen relation will benefit the precision/efficiency trade-off of the widening.

One option is to use semantics preserving congruence relations, i.e., refinements of the congruence relation '$\equiv_{\gamma_{\mathrm{P}}}$'. The availability of relatively efficient but incomplete tests for semantic equivalence can thus be exploited to improve the efficiency and/or the precision of the analysis. As the purpose of this paper is to provide generic widening procedures for powersets that are independent of the underlying domains and hence, of any intended concretizations, here we define these congruences in a way that is independent of the particular concrete domain adopted. Two such relations are the *identity congruence* relation, where no non-trivial equivalence is assumed, and the $\oplus$-*congruence* relation, where sets that have the same join are equivalent. However, the identity congruence will have no influence on the convergence of the iteration sequence, while the $\oplus$-congruence is usually the basis of the default, roughest heuristics for ensuring termination. We now define a new congruence relation that lies between these extremes.

Definition 9. ('$\lhd$' and '$\bowtie$'.) *The* content *relation* $\lhd \subseteq \wp_{\mathrm{fn}}(D,\vdash) \times \wp_{\mathrm{fn}}(D,\vdash)$ *is such that* $S_1 \lhd S_2$ *holds if and only if for all* $S_1' \in \wp_{\mathrm{fn}}(D,\vdash)$ *where* $S_1' \vdash_{\mathrm{P}} S_1$ *there exists* $S_1'' \in \wp_{\mathrm{fn}}(D,\vdash)$ *such that* $\bigoplus S_1' = \bigoplus S_1''$ *and* $S_1'' \vdash_{\mathrm{P}} S_2$. *The* same-content *relation* $\bowtie \subseteq \wp_{\mathrm{fn}}(D,\vdash) \times \wp_{\mathrm{fn}}(D,\vdash)$ *is such that* $S_1 \bowtie S_2$ *holds if and only if* $S_1 \lhd S_2$ *and* $S_2 \lhd S_1$.

Observe that the identity congruence relation can be obtained by strengthening the conditions in the definition of '$\lhd$', replacing $\bigoplus S_1' = \bigoplus S_1''$ with $S_1' = S_1''$; and the $\oplus$-congruence can be obtained by weakening the conditions, replacing $S_1'' \vdash_{\mathrm{P}} S_2$ with $\bigoplus S_1'' \vdash \bigoplus S_2$. Thus the same-content relation is a compromise between keeping all the information provided by the explicit set structure, as done by the identity congruence, and losing all of this information, as occurs with the $\oplus$-congruence.

For the finite powerset domain of polyhedra $(\widehat{\mathbb{CP}}_n)_{\mathrm{P}}$, the content relation '$\lhd$' corresponds to the condition that all the points in polyhedra in the first set are contained by polyhedra in the second set; and hence, the same-content congruence relation '$\bowtie$' coincides with the induced congruence relation '$\equiv_{\gamma_{\mathrm{P}}^{\mathrm{A}}}$'.

Proposition 3. *For all* $\mathcal{S}_1, \mathcal{S}_2 \in \wp_{\mathrm{fn}}(\mathbb{CP}_n, \subseteq)$, $\mathcal{S}_1 \bowtie \mathcal{S}_2$ *if and only if* $\mathcal{S}_1 \equiv_{\gamma_{\mathrm{P}}^{\mathrm{A}}} \mathcal{S}_2$.

Example 7. For $\mathcal{T}_1, \mathcal{T}_2 \in \wp_{\mathrm{fn}}(\mathbb{CP}_1, \subseteq)$ as defined in Example 1, we have $\mathcal{T}_1 \bowtie \mathcal{T}_2$. Consider also $\mathcal{T}_3, \mathcal{T}_4 \in \wp_{\mathrm{fn}}(\mathbb{CP}_1, \subseteq)$ where

$$\mathcal{T}_3 := \big\{\{0 \le x \le 3\}, \{1 \le x \le 5\}\big\}, \qquad \mathcal{T}_4 := \big\{\{0 \le x \le 5\}\big\}.$$

Then $\mathcal{T}_3 \bowtie \mathcal{T}_4$ and also $\mathcal{T}_2 \lhd \mathcal{T}_4$ although the converse does not hold. To see this, let S_1, S_2, and S_2' in Definition 9 be $\mathcal{T}_2$, $\mathcal{T}_4$, and $\mathcal{T}_4' := \big\{\{x = 2.5\}\big\}$, respectively. Then, if $\mathcal{T}_4''$ is such that $\biguplus \mathcal{T}_4'' = \biguplus \mathcal{T}_4'$ and $\mathcal{T}_4'' \subseteq_{\mathrm{P}} \mathcal{T}_4'$, we must have $\mathcal{T}_4'' = \mathcal{T}_4' \not\subseteq_{\mathrm{P}} \mathcal{T}_2$; hence, although $\mathcal{T}_4 = \big\{\biguplus \mathcal{T}_2\big\}$, we have $\mathcal{T}_4 \not\lhd \mathcal{T}_2$.

We now define an operation *merger* that is parametric with respect to the congruence relation and replaces selected subsets by congruent singleton sets.

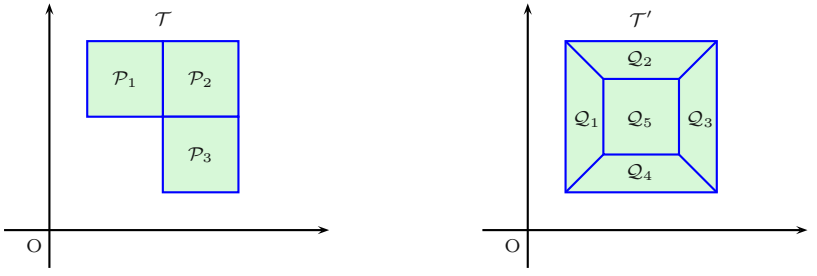


Fig. 1. Merging polyhedra according to '$\bowtie$'.

Definition 10. (Merge and mergers.) *Let R be a congruence relation on $\hat{D}_{\text{P}}$. Then the* merge *relation* $\text{merge}_{_R} \subseteq \wp_{\text{fn}}(D,\vdash) \times \wp_{\text{fn}}(D,\vdash)$ *for R is such that* $\text{merge}_{_R}(S_1, S_2)$ *holds if and only if $S_1 \vdash_{\text{P}} S_2$ and*

$$\forall d_2 \in S_2 : \exists S_1' \subseteq S_1 \,.\, d_2 = \bigoplus S_1' \wedge \{d_2\} \; R \; S_1'.$$

A set $S \in \wp_{\text{fn}}(D,\vdash)$ is fully-merged for R, *if* $\text{merge}_{_R}(S, S')$ *implies $S = S'$; S is* pairwise-merged for R *if, for all $d_1, d_2 \in S$, we have that $\{d_1, d_2\}$ is fully-merged. An operator $\uparrow_R \colon \wp_{\text{fn}}(D,\vdash) \to \wp_{\text{fn}}(D,\vdash)$ is a* merger *for R if* $\text{merge}_{_R}(S, \uparrow_R S)$ *holds.*

Note that, for all $S \in \wp_{\text{fn}}(D,\vdash)$ and congruence relations R, we have $S \vdash_{\text{EM}} \uparrow_R S$.

For the finite powerset domain over $\widehat{\mathbb{CP}}_n$, lines 1–9 of the algorithm specified in [7, Figure 8, page 773] define a merger operator '$\uparrow_{\bowtie}$' such that, for each finite set $\mathcal{S}$ of polyhedra, $\uparrow_{\bowtie} \mathcal{S}$ is pairwise-merged.

Example 8. Figure 1 shows two examples of sets of polyhedra. In the left-hand diagram, the set $\mathcal{T} = \{\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3\}$ of three squares is not pairwise-merged for '$\bowtie$' since $\mathcal{P}_1 \cup \mathcal{P}_2$ and $\mathcal{P}_2 \cup \mathcal{P}_3$ are convex polyhedra. Both $\mathcal{T}_1 = \{\mathcal{P}_1 \cup \mathcal{P}_2, \mathcal{P}_3\}$ and $\mathcal{T}_2 = \{\mathcal{P}_1, \mathcal{P}_2 \cup \mathcal{P}_3\}$ are fully-merged and hence pairwise-merged for '$\bowtie$', and $\text{merge}_{\bowtie}(\mathcal{T}, \mathcal{T}_i)$ holds for $i = 1, 2$. In the right-hand diagram, the set $\mathcal{T}' = \{\mathcal{Q}_1, \mathcal{Q}_2, \mathcal{Q}_3, \mathcal{Q}_4, \mathcal{Q}_5\}$ is pairwise-merged but not fully-merged for '$\bowtie$'. Since $\mathcal{Q}' := \bigcup \mathcal{T}'$ is a convex polyhedron, the singleton set $\{\mathcal{Q}'\}$ is fully-merged and hence pairwise-merged for '$\bowtie$' and $\text{merge}_{\bowtie}\big(\mathcal{T}', \{\mathcal{Q}'\}\big)$ holds.

6 Conclusion

We have studied the problem of endowing any abstract domain obtained by means of the finite powerset construction with a provably correct widening operator. We have proposed two generic widening operators and we have instantiated our techniques, which are completely general, on powersets of convex polyhedra, an abstract domain that is being used for static analysis and abstract model-checking and for which no non-trivial widening operator was previously known.

We have extended the *Parma Polyhedra Library* (PPL) [3,6], a modern C++ library for the manipulation of convex polyhedra, with a prototype implementation of the widenings and their variants employing the 'widening up to' technique [20,21]. The experimental work has just started, but the initial results obtained are very encouraging as our new widenings compare favorably, both in terms of precision and efficiency, with the extrapolation operator of [7].

References

1. S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, chapter 1, pages 1–168. Clarendon Press, Oxford, UK, 1994.
2. R. Bagnara. A hierarchy of constraint systems for data-flow analysis of constraint logic-based languages. *Science of Computer Programming*, 30(1–2):119–155, 1998.
3. R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. *The Parma Polyhedra Library User's Manual*. Department of Mathematics, University of Parma, Parma, Italy, release 0.5 edition, April 2003. Available at `http://www.cs.unipr.it/ppl/`.
4. R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. In R. Cousot, editor, *Static Analysis: Proceedings of the 10th International Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 337–354, San Diego, California, USA, 2003. Springer-Verlag, Berlin.
5. R. Bagnara, P. M. Hill, and E. Zaffanella. Widening operators for powerset domains. Quaderno, Dipartimento di Matematica, Università di Parma, Italy, 2003. To appear.
6. R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill. Possibly not closed convex polyhedra and the Parma Polyhedra Library. In M. V. Hermenegildo and G. Puebla, editors, *Static Analysis: Proceedings of the 9th International Symposium*, volume 2477 of *Lecture Notes in Computer Science*, pages 213–229, Madrid, Spain, 2002. Springer-Verlag, Berlin.
7. T. Bultan, R. Gerber, and W. Pugh. Model-checking concurrent systems with unbounded integer variables: Symbolic representations, approximations, and experimental results. *ACM Transactions on Programming Languages and Systems*, 21(4):747–789, 1999.
8. A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Combinations of abstract domains for logic programming: Open product and generic pattern construction. *Science of Computer Programming*, 38(1–3):27–71, 2000.
9. P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In B. Robinet, editor, *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.
10. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, New York, 1977. ACM Press.
11. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 269–282, New York, 1979. ACM Press.
12. P. Cousot and R. Cousot. Abstract interpretation and applications to logic programs. *Journal of Logic Programming*, 13(2&3):103–179, 1992.

13. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.
14. P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming*, volume 631 of *Lecture Notes in Computer Science*, pages 269–295, Leuven, Belgium, 1992. Springer-Verlag, Berlin.
15. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 84–96, Tucson, Arizona, 1978. ACM Press.
16. G. Delzanno and A. Podelski. Model checking in CLP. In R. Cleaveland, editor, *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99*, volume 1579 of *Lecture Notes in Computer Science*, pages 223–239, Amsterdam, The Netherlands, 1999. Springer-Verlag, Berlin.
17. N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, 1979.
18. G. Filé and F. Ranzato. The powerset operator on abstract interpretations. *Theoretical Computer Science*, 222:77–111, 1999.
19. N. Halbwachs. *Détermination Automatique de Relations Linéaires Vérifiées par les Variables d'un Programme*. Thèse de 3ème cycle d'informatique, Université scientifique et médicale de Grenoble, Grenoble, France, March 1979.
20. N. Halbwachs. Delay analysis in synchronous programs. In C. Courcoubetis, editor, *Computer Aided Verification: Proceedings of the 5th International Conference*, volume 697 of *Lecture Notes in Computer Science*, pages 333–346, Elounda, Greece, 1993. Springer-Verlag, Berlin.
21. N. Halbwachs, Y.-E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, 1997.
22. C. Holzbaur. OFAI clp(q,r) manual, edition 1.3.3. Technical Report TR-95-09, Austrian Research Institute for Artificial Intelligence, Vienna, 1995.
23. W. Kelly, V Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega library interface guide. Technical Report CS-TR-3445, Department of Computer Science, University of Maryland, College Park, MD, USA, 1995.
24. H. Le Verge. A note on Chernikova's algorithm. *Publication interne* 635, IRISA, Campus de Beaulieu, Rennes, France, 1992.
25. V. Loechner. *PolyLib*: A library for manipulating parameterized polyhedra. Available at `http://icps.u-strasbg.fr/~loechner/polylib/`, March 1999. Declares itself to be a continuation of [28].
26. W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, 1992.
27. D. Srivastava. Subsumption and indexing in constraint query languages with linear arithmetic constraints. *Annals of Mathematics and Artificial Intelligence*, 8(3–4):315–343, 1993.
28. D. K. Wilde. A library for doing polyhedral operations. Master's thesis, Oregon State University, Corvallis, Oregon, December 1993. Also published as IRISA *Publication interne* 785, Rennes, France, 1993.

Type Inference for Parameterized Race-Free Java*

Rahul Agarwal and Scott D. Stoller

Computer Science Dept., SUNY at Stony Brook, Stony Brook, NY 11794-4400

Abstract. We study the type system introduced by Boyapati and Rinard in their paper "A Parameterized Type System for Race-Free Java Programs" and try to infer the type annotations ("lock types") needed by their type checker to show that a program is free of race conditions. Boyapati and Rinard automatically generate some of these annotations using default types and static inference of lock types for local variables, but in practice, the programmer still needs to annotate on the order of 1 in every 25 lines of code. We use run-time techniques, based on the lockset algorithm, in conjunction with some static analysis to automatically infer most or all of the annotations.

1 Introduction

Type systems are well established as an effective technique for ensuring at compile-time that programs are free from a wide variety of errors. New type systems are being developed by researchers at an alarming rate. Many of them are very elaborate and expressive.

Types provide valuable compile-time guarantees, but at a cost: the programmer must annotate the program with types. Annotating new code can be a significant burden on programmers. Annotating legacy code is a much greater burden, because of the vast quantity of legacy code, and because a programmer might need to spend a long time studying the legacy code before he or she understands the code well enough to annotate it.

Type inference reduces this burden by automatically determining types for some or all parts of the program. A type inference mechanism is *complete* if it can infer types for all typable programs.

Traditional type inference is based on static analysis. A common approach is constraint-based type inference, which works by constructing a system of constraints (of appropriate forms) that express relationships between the types of different parts of the program and then solving the resulting constraints.

Unfortunately, complete type inference is impossible or infeasible for many expressive type systems. This motivates the development of incomplete type inference algorithms. These algorithms fall on a spectrum that embodies a trade-off

* This work was supported in part by NSF under Grant CCR-9876058 and ONR under Grants N00014-01-1-0109 and N00014-02-1-0363. Authors' Email: {ragarwal,stoller}@cs.sunysb.edu Web: http://www.cs.sunysb.edu/~{ragarwal,stoller}

B. Steffen and G. Levi (Eds.): VMCAI 2004, LNCS 2937, pp. 149–160, 2004.

between computational cost and power. Roughly speaking, we measure an algorithm's power by how many annotations the user must supply in order for the algorithm to succesfully infer the remaining types. For some important type systems, even incomplete algorithms designed to infer most types for most programs encountered in practice may have prohibitive exponential time complexity.

We have developed a new run-time approach to type inference that, for some type systems, appears to be more effective in practice than traditional static type inference. We monitor some executions of the program and infer (one might say "guess") candidate types based on the observed behavior.

A premise underlying this work is that data from a small number of simple executions is sufficient to infer most or all of the types. In particular, it is *not* necessary for the monitored executions collectively to achieve—or even come close to—full statement coverage in order to support successful inference of types for the entire program. Another premise is that the type inference is relatively insensitive to the choice of test inputs. Experience with Daikon [Ern00] supports this idea.

This approach has some obvious theoretical limitations. It is not complete, because the process of generalizing from relationships between specific objects in a particular execution to static relationships between expressions or statements in the program is based in part on (incomplete) heuristics. Run-time type inference is unsound (*i.e.*, the inferred types might not satisfy the type checking rules), because the observed behavior is not necessarily characteristic of all possible behaviors of the program, and correct types express properties that should hold for all possible behaviors of the program. We transform this unsoundness into incompleteness by running the type-checker after type inference. If the type-checker signals an error, we report that type inference failed. This is not a problem in practice, provided it is rare.

Despite these theoretical limitations, from a pragmatic point of view, the most important question for a given type system is whether run-time type inference, traditional static type inference, or a combination of them will provide the greatest overall reduction of users' annotation burden. As a first step towards the empirical evaluation of run-time type inference, we developed and implemented a run-time type inference algorithm for a recently proposed type system for concurrent programs.

Concurrent programs are notorious for containing errors that are difficult to reproduce and diagnose at run-time. This inspired the development of type and effect systems (for brevity, we will call them "type systems" hereafter) that statically ensure the absence of some common kinds of concurrent programming errors. Flanagan and Freund [FF00] developed a type system that ensures that a Java program is race-free, *i.e.*, contains no race conditions; a *race condition* occurs when two threads concurrently access a shared variable and at least one of the accesses is a write. The resulting programming language (*i.e.*, Java with their extensions to the type system) is called *Race Free Java*. Boyapati and Rinard [BR01] modified and extended Flanagan and Freund's type system to make it more expressive. The resulting programming language is called *Param-*

eterized Race Free Java (PRFJ). Hereafter, we assume that programs contain all type information required by the standard Java type-checker, and we use the word "types" to refer only to the *additional* type information required by these extended type systems.

These type systems encode programming patterns that experienced programmers often use to avoid these errors. Specifically, these type systems track the use of locks to "protect" (*i.e.*, prevent concurrent access to) shared data structures. Although these type systems are occasionally undesirably restrictive, experience indicates that they are sufficiently expressive for many programs.

The cost of expressiveness is that type inference for these type systems is difficult. In Flanagan and Freund's experiments with three medium or large programs, well-chosen defaults, in combination with some potentially unsafe (but usually safe in practice) "escapes" from the type system, reduce the number of annotations needed to about 12 annotations/KLOC (KLOC denotes "thousand lines of code") on average [FF00]. For a fair comparison with type inference for PRFJ (discussed below), note that PRFJ programs typically require more annotations than this, primarily because the PRFJ type system is more expressive and does not rely on potentially unsafe escapes.

Flanagan and Freund subsequently developed a simple type inference algorithm for their type system. Roughly speaking, it starts with a set of candidate types for each expression, runs the type checker, deletes some of the candidate types based on the errors (if any) reported by the type checker, and repeats this process until the type checker reports no errors [FF01]. However, this algorithm infers types only for a restricted version of the type system—specifically, a version without external locks. Elimination of external locks significantly reduces the expressiveness of the type system.

Boyapati and Rinard [BR01] use carefully-chosen defaults and local type inference to reduce the annotation burden on users of PRFJ. The user provides type annotations on selected declarations of classes, fields, and methods, and selected object allocation sites (*i.e.*, calls to `new`). Default types are used where annotations are omitted. A simple intra-procedural constraint-based type inference algorithm is used to infer types for local variables of each method. In their experiments with several small programs, users needed to supply about 25 annotations/KLOC [BR01].

We believe that type systems like PRFJ are a promising practical approach to verification of race-freedom for programs that use locks for synchronization, except that the annotation burden is currently too high. We attempted to develop static inter-procedural type inference algorithms for PRFJ, using abstract interpretation [CC77] and constraint-based analysis (*e.g.*, instantiation constraints [AKC02]), but the analysis algorithms were computationally expensive, and we did not believe they would scale to large programs.

This motivated us to develop and implement a run-time type inference algorithm for PRFJ. The program is instrumented by an automatic source-to-source transformation. The instrumented program writes relevant information (mainly information about which locks are held when various objects are accessed) to

a log file. Analysis of the log, together with a simple static program analysis that identifies unique pointers [AKC02], produces type annotations at selected program points. Boyapati and Rinard's simple intra-procedural type inference algorithm is then used to propagate the resulting types to other program points. This has the crucial effect of propagating type information into branches of the program that were not exercised in the monitored executions. Our experience, reported in detail in Section 6, is that run-time type inference provides a significant further reduction in the annotation burden.

If multiple types for (say) a field declaration are consistent with the logged information, we use heuristics to prioritize them. If the candidate type with the highest priority is rejected by the type checker (because it is inconsistent with the types at other program points), we try the next one. In our experiments so far, the highest-priority choice has always worked, so this iterative approach has not been needed (or implemented).

Future Work. In the short term, we plan to implement a type checker for PRFJ; our current lack of a type checker limited our experiments to programs small enough for us to type-check manually. The run-time overhead of our instrumentation is already moderate (see Section 6) but could perhaps be reduced by incorporating ideas from [vPG01]. In the longer term, we are investigating type inference for race-free variants of C, such as [Gro03]. We expect run-time type inference to be even more beneficial in this context. Boyapati and Rinard's defaults in PRFJ are very effective, in part because Java provides built-in locks, so `this` is a very good guess at the identity of the protecting lock in many cases. Devising equally effective defaults for variants of C seems difficult.

2 Related Work

Run-time type inference is very similar in spirit to Daikon [Ern00]. During execution of a program, Daikon evaluates a large syntactic class of predicates at specified program points and determines, for each of those program points, the subset of those predicates that always held at that program point during the monitored executions. Among those predicates, those that satisfy some additional criteria are reported as candidate invariants. Daikon cannot infer PRFJ types, because the invariants expressed by PRFJ types are not expressible in Daikon's language for predicates. Daikon infers predicates that can be evaluated at a single program point. In contrast, a single PRFJ type annotation can express an invariant that applies to many program points. For example, if the declaration of a field f in a class C is annotated with the PRFJ type `self`, it means (roughly): for all instances o of class C, for all objects o' ever stored in field f of o, o' is protected by its own lock, *i.e.*, the built-in lock associated (by the Java language semantics) with o' is held whenever any field of o' is accessed. Such accesses may occur throughout the program.

Naik and Palsberg developed an expressive type system for ensuring that an interrupt-driven program will handle every interrupt within a specified deadline

[NP03]. Their type system is equivalent to model checking, in the sense that a program is typable exactly when their model checker, applied to a specified abstraction of the program, verifies that all deadlines are met. Due to this close connection, types for a program can be inferred from the output of the model checker. Our goal is quite different than theirs: we aim to show that inexpensive run-time techniques (in contrast to relatively expensive model checking) can provide an effective basis for type inference.

Static analyses such as meta-compilation [HCXE02] and type qualifiers [FTA02] have been used to check or verify simple lock-related properties of concurrent programs, *e.g.*, that a lock is not acquired twice by the same thread without an intervening release. Such analyses cannot easily be used to check more difficult properties such as race-freedom.

3 Overview of Parameterized Race Free Java (PRFJ)

The PRFJ type system is based on the concept of object ownership. Each object is associated with an owner which is specified as part of the type of the variables that refer to that object. Each object is owned by another object, or by special values `thisThread`, `self`, `unique` or `readonly`. Since an object can be owned by another object which in turn could be owned by another object, the ownership relation can be regarded as a forest of rooted trees, where the roots may have self loops. Ownership information expresses a synchronization discipline: to safely access an object o, a thread must hold the lock associated with the root r of the ownership tree containing o; r is called o's *root owner*.

An object with root owner `thisThread` is unshared. Such objects can be accessed without synchronization. This is reflected in the type system by declaring that every thread implicitly holds the lock associated with `thisThread`. An object with owner `self` is simply owned by itself. If an object o has owner `unique`, there is a single (unique) reference to o. Only the thread currently holding that reference can access o, so there is no possibility of race conditions involving o, and no lock needs to be held when accessing o. An object with owner `readonly` cannot be updated and can be accessed without any locks.

Every class in PRFJ is parameterized with one or more parameters. Parameterization allows the programmer to specify appropriate ownership information separately for each use of the class. The first parameter always specifies the owner of the `this` object. The remaining parameters, if any, may specify the owners of fields or parameters or return values of methods. The first parameter of a class can be a formal owner parameter or one of the special values discussed above; the remaining parameters must be formal owner parameters. When the class is used in the program, its formal owner parameters are instantiated with final expressions or the above special values. *Final expressions* are expressions whose value does not change; using them to represent owners ensures that an object's owner does not change from one object to another. Syntactically, final expressions are built from final variables, including the implicit `this` variable, final fields, and static final fields. Ownership changes that do not lead to race

```
public class MyThread<thisThread> extends Thread<thisThread> {

  public ArrayList<self,readonly> ls;

  public MyThread(ArrayList<self,readonly> ls) {
    this.ls = ls;
  }

  public void run() {
   synchronized(this.ls) { ls.add(new Integer<readonly>(10)); }
  }

 public static void main(String args[]) {
  ArrayList<self,readonly> ls = new ArrayList<self,readonly>();
  MyThread<unique> m1 = new MyThread<unique>(ls);
  MyThread<unique> m2 = new MyThread<unique>(ls);

  m1--.start();
  m2--.start();
 }
}
```

Fig. 1. A Sample PRFJ Program.

conditions are allowed; for example, an object's owner may change from `unique` to any other owner.

Every method is annotated with a clause of the form "`requires` $e_1, \ldots, e_n$, where the e_i are final expressions. Locks on the root owners of the objects listed in the `requires` clause must be held at each call site.

The type checking rules ensure that in a well-typed program, an object that is not readonly can be accessed only by a thread that either holds the lock on the root owner of the object or has a unique reference to the object. This implies that the program is race-free.

To illustrate the PRFJ system, consider the program in Figure 1. The definition of class `ArrayList` is not shown, but it has two owner parameters: the first specifies the owner of the `ArrayList` itself, and the second specifies the owner of the objects stored in the `ArrayList`. The `MyThread` constructor returns a unique reference to the newly allocated object. Thus, the main thread has unique references to the two instances of `MyThread` until they are started. After an instance of `Mythread` is started, it is accessed by only one thread (namely, itself) and hence is unshared. Thus, the owner of each `MyThread` object changes from `unique` to `thisThread`. The occurrences of `--` in the `main` method indicate that the main thread relinquishes its unique references to `m1` and `m2` when it starts them. These occurrences of `--` are required by the type checking rules, and we consider them to be, in effect, type annotations.

The two instances of `MyThread` share a single `ArrayList` object a. The lock associated with a is held at every access to a, so a has owner `self`, and the first parameter of `ArrayList` is instantiated with `self`. Instances of the `Integer` class are immutable, so they have owner `readonly`. All objects stored in a have owner `readonly`, so the second owner parameter of `ArrayList` is instantiated with `readonly`.

PRFJ defaults are unable to determine the `unique`, `self`, and `readonly` owners used in this program. Our type-inference algorithm described in Section 4 infers all of the types correctly for this program. [AS03] shows in detail how our algorithm works for this program.

4 Type Inference for PRFJ

Our algorithm has three main steps.

First, the static analysis in [AKC02] is used to infer `unique` and `!e` (" not escaping") annotations for fields, method parameters, return values, and local variables (PRFJ's *!e* annotation corresponds to the `lent` annotation in [AKC02]). We use static analysis for this because it is usually adequate and because run-time determination of which objects have unique references would be expensive.

Second, run-time information is used to infer owners for fields, method parameters and return values. Owners in class declarations are inferred next. For a class whose first owner is inferred to be a constant (*i.e.*, anything other than a formal owner parameter), all occurrences of that class in the program are instantiated with that constant as the first owner.

Third, the intra-procedural type inference algorithm in [BR01, Section 7.1] is applied, to infer the types of local variables whose types have not already been determined.

Few classes need multiple owner parameters, and most of the classes that do are library classes, which can be annotated once and re-used, so we do not attempt to infer which classes C need multiple owner parameters or how those parameters should be used in the declarations of fields and methods of C. We assume this information is given. We do try to infer how to instantiate those owner parameters in all uses of C.

4.1 Inferring Unique Owners

The static uniqueness analysis in [AKC02] is a fairly straightforward flow-sensitive context-insensitive inter-procedural data-flow analysis whose running time is linear in the size of the program. For details, see [AKC02].

4.2 Inferring Owners for Fields, Method Parameters, and Return Values

Let x denote a field, method parameter, or method return type with reference type (*i.e.*, not a base type). To infer the owner of x (*i.e.*, the first-owner in the

type of x), we monitor accesses to a set $S(x)$ of objects associated with x. If x is a field of some class C, $S(x)$ contains objects stored in the f field of instances of C. For a method parameter x, $S(x)$ contains arguments passed through that parameter. For a method return type x, $S(x)$ contains objects returned from the method. Let $\mathrm{FE}(x)$ denote the set of final expressions that are syntactically legal (*i.e.*, in scope) at the program point where x is declared.

After an object o is added to $S(x)$, every access (read or write) to o is intercepted and some information is recorded. Specifically, at the end of run-time monitoring, the following information is available for each object o in $S(x)$: $\mathrm{lkSet}(x, o)$, the set of locks that were held at every access to o after o was inserted in $S(x)$ [SBN+97]; $\mathrm{rdOnly}(x, o)$, a boolean that is true iff no field of o was written (updated) after o was inserted in $S(x)$; $\mathrm{shar}(x, o)$, a boolean that is true iff o is "shared", *i.e.*, multiple threads accessed non-final fields of o after o was inserted in $S(x)$; $\mathrm{val}(x, o, e)$, the value of final expression e at an appropriate point for x and o, for each $e \in \mathrm{FE}(x)$. If x is a field, the appropriate point is immediately after the constructor invocation that initialized o. If x is a parameter of a method m, the appropriate point is immediately before calls to m at which o is passed through parameter x. If x is a return type of a method m, the appropriate point is immediately after calls to m at which o is passed as the return value of m.

The owner of x is determined by the first applicable rule below.

1. If the Java type of x is an immutable class (*e.g.*, `String` or `Integer`), then $\mathrm{owner}(x)$=`readonly`.
2. If $(\forall o \in S(x) : \neg\mathrm{shar}(x, o))$, then $\mathrm{owner}(x)$=`thisThread`.
3. If $(\forall o \in S(x) : \mathrm{rdOnly}(x, o))$, then $\mathrm{owner}(x)$=`readonly`.
4. If $(\forall o \in S(x) : o \in \mathrm{lkSet}(x, o))$, then $\mathrm{owner}(x)$=`self`.
5. Let $E(x)$ be the set of final expressions e in $\mathrm{FE}(x)$ such that, for each object o in $S(x)$, $\mathrm{val}(x, o, e)$ is a lock that protects o; that is, $E(x) = \{e \in \mathrm{FE}(x) \mid \forall o \in S(x) : \mathrm{val}(x, o, e) \in \mathrm{lkSet}(x, o)\}$. If $E(x)$ is non-empty, take $\mathrm{owner}(x)$ to be an arbitrary element of $E(x)$.
6. Take $\mathrm{owner}(x)$ to be a formal owner parameter of the class containing x, normally `thisOwner`.[1]

To reduce the run-time overhead, we restrict $S(x)$ to contain only selected objects associated with x. This typically does not affect the inferred types. We currently use the following heuristics to restrict $S(x)$. For a field x with type C, $S(x)$ contains at most one object created at each allocation site for C. For a method parameter or return type x, $S(x)$ contains at most one object per call site of that method. Also, we restrict $\mathrm{FE}(x)$ to contain only the values of final expressions of the form `this` or `this`.f, where f is a final field.

4.3 Inferring Values of Non-first Owner Parameters

If the Java type of x is a class C with multiple owner parameters, for each formal owner parameter P of C other than the first, we need to infer $\mathrm{owner}_P(x)$,

[1] This rule is not needed for the examples in Section 6.

the value with which P should be instantiated for x. Let $S_P(x)$ denote a set of objects o' associated with P for x and such that P denotes the first owner of o'. In particular, for each o in $S(x)$: (1) for each field f of C declared with P as the first owner of f (*i.e.*, `class` C`<...,`P`,...> { ... D<P> f; ... }`), objects stored in $o.f$ are added to $S_P(x)$; (2) for each parameter p of a method m of C such that the first owner of p is P (*i.e.*, `class` C`<...,`P`,...> { ... m(...,D<P> p,...) ... }`), add to $S_P(x)$ arguments passed through parameter p when $o.m$ is invoked; (3) for each method m of C whose return type has first owner P, add to $S_P(x)$ objects returned from invocations of $o.m$. We instrument the program to monitor accesses to objects in $S_P(x)$ and infer an owner based on that, just as in Section 4.2. As an optimization, we may restrict $S_P(x)$ to contain a subset of the objects described above.

4.4 Inferring Owners in Class Declarations

Let owner(C) denote the first owner in the declaration of class C (*i.e.*, it denotes o in `class` C`<`o`,...>`). Let $S(C)$ contain instances of C. We monitor accesses to elements of $S(C)$ as in Section 4.2 and then use the following rules to determine owner(C).

1. If C is a subclass of a class C' with owner(C')=`self`, then owner(C)=`self`.
2. If $S(C) = \emptyset$ (*i.e.*, there are no instances of C), then owner(C)=`thisOwner`.[2]
3. If $(\forall o \in S(C) : \neg\text{shar}(x, o))$, then owner($C$)=`thisThread`.
4. If $(\forall o \in S : o \in \text{lkSet}(x, o))$, then owner($C$)=`self`.
5. Take owner(C)=`thisOwner`.

For efficiency, we restrict $S(C)$ to contain only a few instances of C. Currently, we arbitrarily pick two fields or method parameters or return values of type C and take $S(C)$ to contain the objects stored in or passed through them.

4.5 Inferring `requires` Clauses

We infer `requires` clauses basically as in [BR01], except we use run-time monitoring instead of user input to determine which classes C have owner `thisThread` (*i.e.*, each instance of C is accessed by a single thread).

Each method declared in each class with owner `thisThread` is given an empty `requires` clause. For each method in each other class, the `requires` clause contains all method parameters p (including the implicit `this` parameter) such that m contains a field access $p.f$ (for some field f) outside the scope of a `synchronized(`p`)` statement; as an exception, the `run()` method of classes that implement `Runnable` is given an empty `requires` clause, because a new thread holds no locks.

[2] For example, in many programs, the class containing the `main` method is never instantiated.

4.6 Static Intra-procedural Type Inference

The last step is to infer the types of local variables whose types have not already been determined, using the intra-procedural type inference algorithm in [BR01, Section 7.1]. Each incomplete type (*i.e.*, each type for which the values of some owner parameters are undetermined) is filled out with an appropriate number of fresh distinct owner parameters. Equality constraints between owners are constructed in a straightforward way from each assignment statement and method invocation. The constraints are solved in almost linear time using the standard union-find algorithm. For each of the resulting equivalence classes E, if E contains one known owner o (*i.e.*, an owner other than the fresh parameters), then replace the fresh owner parameters in E with o. If E contains multiple known owners, then report failure. If E contains only fresh owner parameters, then replace them with `thisThread`. This heuristic is adequate for the examples we have seen, but if necessary, we could instrument the program to obtain runtime information about objects stored in those local variables, and then infer their owners as in Section 4.2.

5 Implementation

This section describes the source-to-source transformation that instruments a program so that it will record the information needed by the type inference algrithm in Section 4. The transformation is parameterized by the set of classes for which types should be inferred.

All instances of `Thread` are replaced with `ThreadwithLockSet`, a new class that extends `Thread` and declares a field `locksHeld`. Synchronized statements and synchronized methods are instrumented to update `locksHeld` appropriately; a `try/finally` statement is used in the instrumentation to ensure that exceptions are handled correctly. For each field, method parameter and return type x being monitored, a distinct `IdentityHashMap` is added to the source code. The hashmap for x is a map from objects o in $S(x)$ to the information recorded for o, as described in Section 4, except the lockset. We store all locksets in a single `IdentityHashMap`. Thus, even if an object o appears in $S(x)$ for multiple x, we maintain a single lockset for o. Object allocation sites, method invocation sites, and field accesses are instrumented to update the hashmaps appropriately. Which sites and expressions are instrumented depends on the set of classes specified by the user.

6 Experience

To evaluate the inference engine, we ran our system on the five multi-threaded server programs used in [BR01]. C. Boyapati kindly sent us the annotated PRFJ code for these servers. The programs are small, ranging from about 100 to 600 lines of code. We compare the number of annotations in that code—this is also the number of annotations needed for the mechanisms in [BR01] to successfully

infer the remaining types—with the number of annotations needed with our type-inference algorithm. Recall that our type-inference algorithm is also incomplete and hence might be unable to infer some types. In these experiments, we infer types only for the application (*i.e.*, server) code; we assume PRFJ types are given for Java API classes used by the servers.

In summary, our type-inference mechanism successfully inferred complete and correct typings for all five server programs, with no user-supplied annotations. Also, slowdown due to the instrumentations was typically about 20% or less.

Four of the server programs did not come with clients, so we wrote very simple clients for them. One server program (PhoneServer) came with a simple client. We modified the servers slightly so they terminate after processing one or two requests (in our current implementation, termination triggers writing of collected information to the log). A single execution of each server with its simple client provided enough information for successful run-time type inference. This supports our conjecture (in Section 1) that data from a small number of simple executions is sufficient.

The original PRFJ code for GameServer requires 7 annotations. Our algorithm infers types for the program with no annotations. We wrote a simple client with two threads. This program is sensitive to the scheduling of threads. Different interleavings of the threads caused different branches to be taken, leaving other branches in the code unexercised in that execution. We applied our run-time type inference algorithm to each of the possible executions, and it sucessfully inferred the types in every case.

The original ChatServer code contains 13 annotations. We wrote a simple client with two threads. One class (`read_from_connection`) in the server program has only final fields, and the class declaration can correctly be typed with owner `self` or `thisThread`. The original code contains a manual annotation of `self`, while our algorithm infers `thisThread`. The ChatServer, like the other servers, uses Boyapati's modified versions of Java API classes (*e.g.*, `Vector`), from which synchronization has been removed. The benefit is that synchronization can be omitted in contexts where it is not needed; the downside is that, when synchronization is necessary, it must be included explicitly in the application code. We also considered a variant of this server that uses unmodified Java library classes. Our algorithm infers a complete and correct typing for both variants, with no assistance from the user.

The original QuoteServer code contains 15 annotations, PhoneServer code contains 12 annotations across 4 classes and HTTPServer contains 23 annotations across 7 classes. Our algorithm infers types for each of these programs with no annotations. Most of the annotations in QuoteServer were unique annotations, and static analysis of uniqueness is able to infer the annotations. For the HTTPServer program the inferred types are not exactly the same as those in the original code, as was the case with `ChatServer`.

Encouraged by these initial results, we plan to apply our system to much larger examples as soon as we have implemented a type-checker for PRFJ.

Acknowledgement. We thank Chandra Boyapati for many helpful comments about PRFJ.

References

[AKC02] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *Proc. 17th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM Press, 2002.

[AS03] Rahul Agarwal and Scott D. Stoller. Type inference for parameterized race-free Java. Technical Report DAR 03–10, Computer Science Department, SUNY at Stony Brook, October 2003.

[BR01] Chandrasekar Boyapati and Martin C. Rinard. A parameterized type system for race-free Java programs. In *Proc. 16th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, volume 36(11) of *SIGPLAN Notices*, pages 56–69. ACM Press, 2001.

[CC77] Patrick Cousot and Radhia Cousot. A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[Ern00] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington, Department of Computer Science and Engineering, 2000.

[FF00] Cormac Flanagan and Stephen Freund. Type-based race detection for Java. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 219–232. ACM Press, 2000.

[FF01] Cormac Flanagan and Stephen Freund. Detecting race conditions in large programs. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 90–96. ACM Press, June 2001.

[FTA02] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 2002.

[Gro03] Dan Grossman. Type-safe multithreading in Cyclone. In *Proc. ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI)*, pages 13–25. ACM Press, 2003.

[HCXE02] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 69–82. ACM Press, 2002.

[NP03] Mayur Naik and Jens Palsberg. A type system equivalent to a model checker. Master's thesis, Purdue University, 2003.

[SBN+97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.

[vPG01] Christoph von Praun and Thomas R. Gross. Object race detection. In *Proc. 16th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, volume 36(11) of *SIGPLAN Notices*, pages 70–82. ACM Press, October 2001.

Certifying Temporal Properties for Compiled C Programs

Songtao Xia and James Hook

OGI School of Science & Engineering
Oregon Health & Science University
Beaverton OR 97006, USA

Abstract. We investigate the certification of temporal properties of untrusted code. This kind of certification has many potential applications, including high confidence extension of operating system kernels. The size of a traditional, proof-based certificate tends to expand drastically because of the state explosion problem. Abstraction-carrying Code (ACC) obtains smaller certificates at the expense of an increased verification time. In addition, a single ACC certificate may be used to certify multiple properties. ACC uses an abstract interpretation of the mobile program as a certificate. A client receiving the code and the certificate will first validate the abstraction and then run a model checker to verify the temporal property.

We have developed ACCEPT/C, a certifier of reachability properties for an intermediate language program compiled from C source code, demonstrating the practicality of ACC. Novel aspects of our implementation include: 1) the use of a Boolean program as a certificate; 2) the preservation of Boolean program abstraction during compilation; 3) the encoding of the Boolean program as program assertions in the intermediate program; and 4) the semantics-based validation of the Boolean program via a verification condition generator (VCGen). Our experience of applying ACCEPT/C to real programs, including Linux and NT drivers, shows a significant reduction in certificate size compared to other techniques of similar expressive power; the time spent on model checking is reasonable.

1 Introduction

Proof-carrying code (PCC) techniques address the problem of how a client trusts a mobile (typically low-level) program by verifying a mathematical proof for certain safety properties [18]. Specifically, a server applies a safety policy to the program to generate verification conditions (VCs) and certifies the safety property with a checkable proof of the VCs. When a client generates the same set of VCs and checks the proof, PCC guarantees the client the safety of that program. Early variants of PCC, including TAL [17,16], ECC [13], Certified Binary [25], FPCC [1], TouchStone [18] and others, focus primarily on the verification of type safety and memory safety, properties easily expressed in first-order logic.

This paper investigates the certification of general temporal properties. Temporal logic [14] can express powerful safety policies, such as the correct use of APIs [2,8], or liveness requirements. Such properties enable an operating system kernel to start a

B. Steffen and G. Levi (Eds.): VMCAI 2004, LNCS 2937, pp. 161–174, 2004.

service developed by an untrusted party with high confidence that critical invariants, such as correct locking discipline, will not be compromised.

Other researchers are also extending the PCC framework to address temporal properties. In most of the proposed systems, a certificate is, as in PCC, a proof of the temporal property. Whether this proof is worked out using a theorem prover, as in the case of Temporal PCC (TPCC) [4], or translated from the computation of a model checker [26, 11,12], the size of the proof tends to be large due to state explosion. In some cases a proof is several times larger than the program size even for trivial properties [4].

We are exploring Abstraction-carrying code (ACC), where an abstract interpretation of the program is sent as a certificate. Our specific strategy is to adopt refinement based [5,20] predicate abstraction techniques [2,21,7,6,3] as the certification method. The abstraction model that is successfully model checked certifies the property of concern. In this paper, we use Boolean programs [2] as the abstract models.

Boolean programs are often computed for source programs. To certify mobile programs communicated in an intermediate language, we compile the source code into the intermediate language while preserving the correspondence with the abstract model. The intermediate program, the predicates and the abstract model are sent to the client. The client generates verification conditions to validate the abstract model and then model checks the abstract model to verify the temporal property.

To investigate the feasibility of ACC, we have constructed the ACC Evaluation Prototype Toolkit for C (ACCEPT/C). This toolkit, while not yet a full implementation of ACC, contains a certifying compiler. ACCEPT/C enables measurement of the size of certificates and the computational resources for certificate generation, validation and re-verification. ACCEPT/C is built on top of BLAST [8] and CIL [15]. The implementation exploits existing capabilities of BLAST and handles additional technical issues. In particular:

- ACCEPT/C computes a Boolean program from BLAST's intermediate result, where a search tree is generated without an explicit concept of a static abstract model that resembles a Boolean program.
- ACCEPT/C compiles the Control Flow Automaton (CFA) representation of the source code to an intermediate program for which the Boolean program is still a valid abstraction.
- ACCEPT/C encodes the Boolean program as program assertions in the compiled program.
- On the client side, the verification condition generator (VCGen) produces a set of conditions that can be discharged by an automatic theorem prover.

Figure 1 illustrates the relationship between major data artifacts used by ACCEPT/C.

To evaluate ACC, we are applying ACCEPT/C to Linux and NT drivers and other public domain programs. For API-compliance properties, most test cases yield a small certificate, acceptable model validation time and a surprisingly small model checking overhead. Initial investigation of a public domain implementation of an SSH server supports our conjecture that a single, yet small, ACC certificate can establish multiple properties.

An alternative approach is the Model-carrying Code (MCC) framework [23,24]. In MCC the model is generated using statistical techniques based on server side observa-

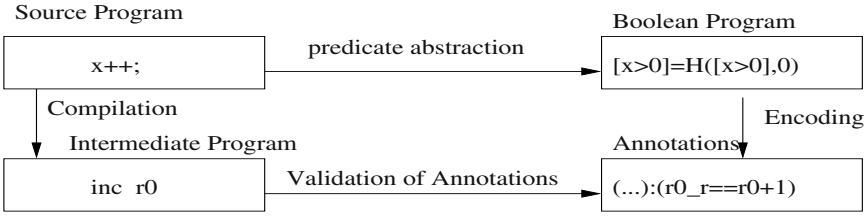


Fig. 1. System overview of ACCEPT/C

tions. The model is validated at run time by the client, though the verification of property is done statically. It does not require the client and the server to share details of the security property being verified. However, without such knowledge, it is sometimes hard to compute a model that is good enough to verify complicated properties. And the dynamic validation of the model introduces further workload for a client.

The paper is organized as follows. In Section 2, we introduce the technical background of ACC. Section 3 introduces how we compute Boolean programs from BLAST's lazy abstraction results. Section 4 presents the abstraction-preserving compilation framework. Section 5 describes the encoding of Boolean programs as program assertions. Section 6 introduces the verification condition generator (VCGen). Section 7 reports our case studies with ACCEPT/C. We assume the readers have some knowledge of predicate abstraction. One example drawn from SLAM is used throughout the paper.

2 Background

This section reviews control flow automata (CFA), a mathematical abstraction used in BLAST [8], and adapts the formalism to models of intermediate representations. These results establish that ACC is expressive over next operator-free Linear Temporal Logic (LTL) formulas built over predicates on the system state. Preliminary accounts of these results are in our earlier workshop papers on ACC [29,28].

2.1 Control Flow Automata

Henzinger and colleagues in the BLAST (Berkeley Lazy Abstraction Software Verification Tool) group use Control Flow Automata (CFA) to encode control flow graph information in a manner suitable for model checking. In particular, the CFA framework provides a mechanism in which computations can be seen as paths in trees accepted by CFAs. The BLAST group discuss CFAs in several publications, where they are motivated by examples [9,10] and presented formally [8].

The CFA formalism characterizes a computation by a control state (or location) and a data state. For every function in the C program, a CFA is built. The set of control states is explicitly represented by a finite control flow graph. Edges in the graph are labeled by symbolic terms that characterize either changes to the data state (basic blocks of C statements) or predicates on the data state that enable the change of control state to

happen (e.g. predicates computed from `if` statement conditionals). Later we will extend a CFA to allow the labels to be basic blocks of an intermediate language.

A set of CFAs representing a program induces a transition system, which accepts an infinite tree of (location, data state) pairs. A stack state is implicit in the path from the root to any node in the tree. Each edge in the tree must correspond to a labeled transition in the CFA and the data state must satisfy the label on the CFA transition. The data state is specified abstractly so that the data states on a path in the tree accepted by the CFA are an abstraction of the concrete states reached during an execution of the program. The abstraction is selected so that the concrete states that result in the same control behavior are identified (states identified in this way are called a *region*). The tree represents the reachable regions of the computation.

Model checking techniques allow this transition system to be tested for compliance with properties expressed in LTL, provided the LTL formula in question is built over atomic formulas that can be tested algorithmically on the CFA state.

The relationship between the abstract data states of the execution tree and the regions of concrete states of the C program can be formulated as a Galois connection, (α, γ). The abstraction function α maps a concrete region to an abstract one. The pair induces an abstract transition relation $\rightarrow_a$ between abstract regions. An abstract transition system can be computed from the abstract state space and $\rightarrow_a$. The abstract transition system can be model checked to verify the property on the concrete system. Because of the Galois connection requirement, the relation $\rightarrow_a$ is an over approximation of the transition relation of the concrete system. As a result, if we model check the abstract transition system, only global properties, such as LTL formulas, can be verified on the original system.

ACCEPT/C adopts Boolean abstraction, where the abstract transition $\rightarrow_a$ is defined by a Boolean program[2].

2.2 Lazy Abstraction

Traditional predicate abstraction builds an abstract model of a program based on a fixed set of predicates, called the abstraction basis. We will call such systems *uniform* because they use the same abstraction basis for every statement in the source program. Traditional approaches are also *static* because they construct the model completely prior to model checking.

Lazy abstraction is a *dynamic, non-uniform* abstraction technique. It builds a model on-the-fly during model checking. The abstraction function is different in different parts of the model checking search tree. Predicates are added only in those portions of the search tree where they are needed. A complete treatment of lazy abstraction can be found in the BLAST literature [8, Section 3].

2.3 Abstracting Intermediate Programs

BLAST models source programs. To support mobile code verification we must develop models of the intermediate code supplied from server to client. There are essentially two ways to approach this: directly build a model of the intermediate program or calculate

a refinement of a model of the source program in parallel with compilation. We have chosen the second approach because significantly more predicates are required at the intermediate level than at the source level to capture an abstract transition relation that enables successful model checking. In addition, to the extent that any tools exist to allow the programmer to assist in predicate discovery and refinement, these tools focus on source level abstractions.

To retain the validity of the abstract model as we compile, we restrict the compilation process to preserve as much of the original control flow behavior of the program as possible. In particular, the intermediate program shares (more precisely, refines) the same control flow graph. This allows us to find a simple correspondence between the compiled program and the Boolean program. Namely, an intermediate block compiled from a source statement is associated with the Boolean program statements corresponding to the source statement. The compilation of an `if` statement is associated with the `assume` statements corresponding to the `if` statement.

In addition to preservation of control flow it is also critical to preserve the semantics of all program values that are necessary to validate the correspondence between the compiled code and the Boolean program. This is accomplished by identifying a set of observable values in the source program and insuring that all residual representations of these values in the intermediate program are explicitly stored in memory. And when a variable is modified, the change should be written through to its memory copy.

Finally, compilation must respect the granularity of the semantics of the temporal logic modalities on the source program. Basic blocks in the intermediate program can be no finer than the unit of observation in the semantics of the temporal modalities. A sufficient condition to achieve this is to restrict basic blocks in the intermediate program to modify at most one variable.

3 Generation of Boolean Programs

This section describes the calculation of the Boolean program that will ultimately be the certificate of the mobile code. The Boolean program will be compact and in many ways directly reflect the structure of the source program. In particular, every program point in the Boolean program will correspond to a single program location in the source program (and the CFA). To construct a Boolean program that can be successfully model checked, we must have some sufficiently refined sets of predicates. In ACCEPT/C, we obtain these predicates sets from the BLAST's lazy abstraction result.

With BLAST, the abstract model generated in lazy abstraction is dynamic and non-uniform. A Boolean program on the other hand is static. We will use a static, non-uniform Boolean program to approximate the dynamic, non-uniform model generated by BLAST to benefit from the strength of both approaches.

BLAST associates a predicate set with each tree edge. To build the Boolean program, predicate sets on all the edges corresponding to the same transition in the CFA must be collected. For an edge in the source CFA, the predicates used to construct a Boolean program must be logically equivalent to the union of the predicate sets of the tree edges representing the corresponding transition of the CFA. We then apply the method described in SLAM [2] to compute the Boolean program. For example, for an assignment

statement in C, we may compute an approximation of the weakest precondition of a predicate as the condition for the corresponding Boolean variable to be true.

The proposition below characterizes that the static Boolean program generated is still as appropriate a model of the original program as the dynamic model obtained by lazy abstraction, provided the lazy abstraction used Boolean abstraction locally. That is, we show that if the lazy abstraction-based model checking does not reach an error state, neither will the model checking of the Boolean program constructed above.

A state in lazy abstraction or in a Boolean program can be viewed as a bit vector. Assume the order of the bits are fixed, that is, the same bit in different states corresponds to the same predicate. We say state s refines state t if s agrees with t on every bit of t.

Proposition 1 *If state s_1 refines state s_2, $s_1 \hookrightarrow_1 s_1'$, and $s_2 \hookrightarrow_2 s_2'$, where $\hookrightarrow_1$ is an abstract transition relation induced by a subset of the predicates that induces abstract transition relation $\hookrightarrow_2$, then s_1' refines to s_2'.*

Therefore, if none of the leaves on the search tree contain an error state, neither will any reachable state generated by model checking the Boolean program.

These results show that ACC does not weaken any significant properties of the models we construct. In Figure 2, we list the C code used in the SLAM papers. The topology of the corresponding control flow automaton is illustrated on the right side of Figure 2 (labels are compiled code). According to the SLAM paper, model checking of the correct use of locks is successful only after we can observe the predicate `nPacket` $=$ `nPacketOld`. BLAST will find this predicate and use it to abstract the program only when necessary. In this case it will note that this predicate is only required within the loop. Therefore, we abstract the statements in the loop with three predicates and abstract the rest of the program with two predicates. The corresponding Boolean program is shown to the right.

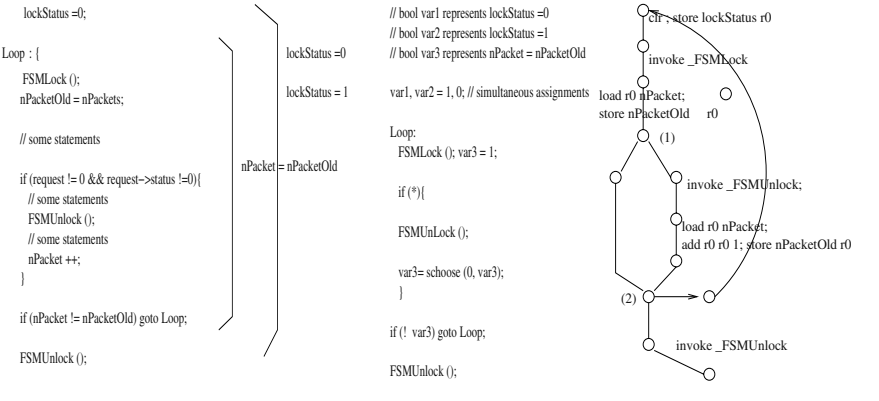


Fig. 2. Correct Use of Locks: NT Driver

4 Abstraction-Preserving Compilation

The abstraction-preserving compilation process starts with the CFA representation of the source program and the Boolean program computed in the previous section. The compilation process must produce an intermediate representation of the source program annotated in such a manner that a Boolean program corresponding to the intermediate representation can be extracted by the client and its fidelity verified. This is accomplished by annotating basic blocks in the intermediate code with simultaneous assignments to the Boolean variables and annotating certain program points, usually the beginning of an instruction sequence for a `then` or `else` clause, with predicates from the Boolean program `assume` statements.

We present the straightforward approach that compiles every basic block in isolation, and maintains the Boolean program essentially unmodified. An extension allowing optimization is explored in an earlier paper [29].

Each source function is characterized by a CFA and a corresponding function in the Boolean program. Compilation is driven by traversing the control graph in the CFA. CFAs have two kinds of edges, those labeled by a basic block and those labeled by an `assume` predicate. For edges labeled by predicates the compilation process emits an appropriate test instruction and branch construct. Such an edge is labeled by the predicates in the corresponding `assume` statement in the Boolean program. For a basic block edge, the basic block is compiled in isolation and annotated with the residual simultaneous assignment to Boolean variables found in the Boolean program. The basic blocks are assembled according to the control flow graph. When multiple paths merge it is sometimes necessary to introduce labels for intermediate proxy nodes that can be labeled with appropriate predicates.

The compilation of our example to an intermediate language is illustrated in Figure 2. The tests of conditions ((1) and (2)) in the intermediate program are not associated with an edge. Instead, they are associated with a node that has multiple out-edges. The direction of edges in the figure is from top to bottom unless explicitly indicated.

One of the important issues is the preservation of the property of concern. This meta-property derives from the coherence of the semantics of the temporal logic, source language and intermediate language. The result is given in detail in an earlier paper [28].

5 Annotation Language

The annotation language is a quantifier-free predicate logic extended with equality, uninterpreted function symbols, simple arithmetic and pointer dereference. The syntax is listed in Figure 3. Symbol c is for constants; v is for variables. Variables in the annotation language represent values in registers and memory. Symbol $\&$ takes the address of its operand. A Boolean program assignment assigns to a Boolean variable b a `schoose` expression, which is evaluated to true if the first argument is true, or false if the second argument is true, or * (non-deterministic choice) when neither argument is true.

There are two forms of annotations. A set of Boolean assignments annotates an intermediate program block compiled from a source statement. The second form of annotation associates a program location with a predicate, representing a program assertion. For

$$
\begin{array}{ll}
\texttt{expressions:} & E ::= c \mid v \mid f(E) \mid E_1 + E_2 \mid \&E \mid \cdots \\
\texttt{predicate:} & P ::= \texttt{true} \mid \neg P \mid P_1 \vee P_2 \mid \cdots \\
 & \quad \mid E_1 = E_2 \\
\texttt{Boolean assignment} & A ::= b = \texttt{schoose}(E_1, E_2) \\
\texttt{Boolean assumption} & ::= \texttt{assume}(P)
\end{array}
$$

Fig. 3. Syntax of the Annotation Language

example, at the beginning of a block compiled from a C `then` clause of an `if` statement we may specify such an assertion. This assertion corresponds to an `assume` statement in a Boolean program. As we will see later, this form of annotation can be used to specify other forms of assertions.

The reason for having pointer dereference is to handle aliases. With the presence of pointers, the computation of a (sound) Boolean program from a C program is expensive because every possible alias condition needs to be addressed; the resulting Boolean program tends to be unnecessarily large. Boolean abstraction on C often adopts pointer analysis to remove impossible cases. For example, if we have an assignment $x := 0$ and predicate $*p = 0$, the assignment may make the predicate true. If alias analysis can guarantee that p and x are not related, then in computing of the Boolean program, we do not have to handle the predicate $*p = 0$. For the alias information to be trusted by a client, we need to put assertions at the beginning of the block, indicating $\neg(\&x = p)$, indicating that at that program point, we know p is not pointing to x. We will use the second form of the annotation mentioned above. Verification of such assertions is the same as verifying an assertion introduced by an `assume` statement.

6 Generating Verification Conditions

We send the compiled program, the predicate sets and the annotations to a client. It is the client's responsibility to validate the annotations. The client will generate VCs and discharge them by calling a theorem prover. This is different than traditional PCC. A PCC system usually generates these VCs on the server side; a server will invoke a certifying theorem prover [19] to generate proofs for these VCs. The proofs will be attached to the mobile program as part of the certificate. A client will verify the proof to discharge the VCs. In ACCEPT/C, we choose not to generate and attach the proof for the VCs to minimize the certificate size. As shown in Section 7, the overhead of theorem prover calls in validating the Boolean program is typically small in practice.

The VCGen applies the semantics of the intermediate language to work out the VCs. Specifically, the intermediate language program defines a post operator `post`. For a block B annotated by assignments of the form $b_v := \texttt{schoose}(e_1, e_2)$ in the Boolean program, we need to validate that e_1 and e_2 are the preconditions of the predicates b_v and $\neg b_v$ represent (which we call e_v and $\neg e_v$). To do so, we compute the post conditions of e_1 and e_2 and check whether they imply e_v and $\neg e_v$, respectively. Note we do not have to verify that e_1 or e_2 is a best approximation of the weakest precondition. Verifying

that they are preconditions will be enough. Formally, we generate a set of verification conditions of the form:

$$(\texttt{post}(c \wedge e_1, B) \Rightarrow e_v) \wedge (\texttt{post}(c \wedge e_2, B) \Rightarrow \neg e_v) \quad (1)$$

where c is true if B has no location annotation, or P if the beginning of B is annotated by an assertion (P).

Consider a block B which is a test and jump. This block must be compiled from an `if` statement in C. There must be an assertion specified at the target of the jump, which we call P_1. If B_j is the instruction sequence that computes the test condition, we generate the verification condition: $\texttt{post}(c, B_j) \wedge P_C \Rightarrow P$, where c is as defined above and P_C is the test condition. This VC verifies the encoded `assume` statement or any other assertions annotated at the beginning of the target. If there is an `else` branch, we also generate a VC to verify the assertion specified at the next block, if any.

Below, we show how to combine the principles above with the semantics of the intermediate language to generate verification conditions. We first present the semantics and then describe how to compute the post condition. In the two subsections below, we take a small portion of the intermediate language similar to the one used in ACCEPT/C to demonstrate our techniques.

6.1 Semantics of Intermediate Language

We present a simplified version of semantics of the intermediate language, which is running on an abstract machine. The abstract machine assumes a number (n) of registers and unlimited memory. The machine state in this subsection is a pair $(\texttt{ic}, M)$, where `ic` is the instruction counter and M maps registers and variables to integers. For example, $M(i)$ represents the i-th register and $M(x)$ represents the value of x. Let u range over registers and variables. We write $M[u \mapsto v]$ for a mapping where $M(u)$ is updated to v. We assume that `ic` are integers and that $J(l)$ returns the instruction at location l.

We present a selected set of instructions. These instructions and their semantics are listed in Figure 4.

Instruction	Next PC	Effect	Transition Condition
`add` $r_a\ r_b\ r_i$	$\texttt{ic} + 1$	$M(i) := M(a) + M(b)$	
`store` r_i `var`	$\texttt{ic} + 1$	$M(\texttt{var}) := M(i)$	
`jeq` $l\ r_i\ r_j$	$\texttt{ic} + 1$		$M(i)! = M(j)$
	l		$M(i) = M(j)$

Fig. 4. Selected Semantic Rules

For example, an `add` instruction adds two operands and puts the result in the result location. A `store` instruction takes a register and a variable and stores the content of the register in the variable. A test and jump instruction `jeq` jumps to different program locations according to the test.

6.2 VCGen

Intuitively, generation of the verification conditions has two steps. First, we compute the post condition of a block of instructions. Then we compare the post condition with the annotations, generating a verification condition.

In Figure 5 we list selected rules for computing the post condition. A post condition is considered as an abstract machine state. The abstract machine state used in this subsection is defined by a formula in the same logic in which we write annotations. We use σ for such an abstract state and v_u for the value of u. For example, the rule for instruction `add` says, for an input abstract machine state σ, the next state is different from σ in that the value of location i is the sum. Expression $\sigma[v_i \mapsto v_x + v_y]$ can be defined as $(\exists v_i.\sigma) \wedge (v_i = v_x + v_y)$. The old value of i is quantified. In practice this is computed by substituting v_i with a fresh variable unless v_i appears in the annotations[27].

The VCGen scans a block at a time. When the end of the block is reached, we will test to see whether the annotations specified for the block hold. Using (1), we generate verification conditions that validate the annotated Boolean assignments. We rename the variables if necessary because some variables in (1) are from the starting abstract state and some are from the resulting abstract state. As a convention, a register or variable subscripted with an "r" refers to the value from the final abstract state.

For example, the instruction sequence for a C statement $\texttt{nPacket}++$ is:

$$\texttt{load}\ r_0\ \texttt{nPacket}; \texttt{add}\ r_0\ r_0\ 1; \texttt{store}\ r_0\ \texttt{nPacket}$$

The Boolean program of concern is $\texttt{var3} := \texttt{schoose}(0, \texttt{var3})$. In the logic, the Boolean statement is encoded as $\texttt{var3} \Rightarrow \neg\texttt{var3}_r$, where `var3` is $\texttt{nPacket} = \texttt{nPacketOld}$. This encoded annotation reads, "if $\texttt{nPacket} = \texttt{nPacketOld}$ in the starting abstract state, then $\texttt{nPacket} = \texttt{nPacketOld}$ is not true in the final abstract state". The VCGen process can be intuitively interpreted as: From an initial state σ where $\texttt{nPacket} = \texttt{nPacketOld}$, we compute the post state of the instruction sequence, which will be described as formulas: $r_{0r} = \texttt{nPacket} + 1; \texttt{nPacket}_r = r_{0r}$. This formula is tested to see if it implies

$$(\texttt{nPacket} = \texttt{nPacketOld} \Rightarrow \neg\texttt{nPacket}_r = \texttt{nPacketOld}_r)$$

Results from our earlier work [29] prove that these steps validate the encoded Boolean program.

We also generate VCs to validate assertions. Specifically, meta-operation *test* `inv` generates an implication between the current abstract state we have computed and the assertion `inv` to be verified, handling necessary substitution. Meta function I maps a location to the assertions specified at that location, or true if none is specified. Also, because the result of pointer analysis is encoded as state assertions in the same logic, the verification conditions generated will verify these data flow facts as well. For example, for instruction `jeq`, we add the test condition to the current abstract state to verify the assertions specified at the transition target; we add the negation of the test condition to the abstract state to verify the assertions specified at the next program location.

Instruction	$InputState$	$NextState$	VC
`add`	σ	$\sigma[v_i \mapsto v_x + v_y]$	`test` $I(\texttt{pc}+1)$
`store` r_i `x`	σ	$\sigma[v_x \mapsto v_i]$	`test` $I(\texttt{pc}+1)$
`iaload` $r_i\ r_j\ r_k$	σ	$\sigma[v_i \mapsto *(v_j + v_k)]$	`test` $I(\texttt{pc}+1)$ `memsafe`(`access`(r_j, r_k))
`jeq` $r_i\ r_j$	σ	$\sigma \wedge (v_i = v_j)$	`test` $I(l)$
		$\sigma \wedge \neg(v_i = v_j)$	`test` $I(\texttt{pc}+1)$

Fig. 5. Post Condition Computation and VC Generation

7 Experience

To test the feasibility of ACC, we have constructed the ACC Evaluation Prototype Toolkit for C (ACCEPT/C). The initial goals for ACCEPT/C, besides as a prototype implementation of a certifying compiler, are to support critical measurements to determine the feasibility of the ACC approach. These measurements include the computational resources required to generate, validate and model check a certificate. Ultimately, we expect to develop a full ACC implementation based on ACCEPT/C.

ACCEPT/C builds directly on the BLAST implementation from Berkeley, which incorporates the CIL infrastructure for C language programs. BLAST directly gives a mechanism to compile C to a CFA and to apply lazy abstraction to find a set of predicates suitable for model checking properties of C programs. We have modified this portion to measure model validation time.

ACCEPT/C extends BLAST with the ability to recover a Boolean Program from the CFA and the predicate set used in BLAST's lazy abstraction. This capability allows ACCEPT/C to generate models that can be checked by other model checking tools. In particular, we have implemented an interface to Moped [22].

ACCEPT/C also extends BLAST with a compilation mechanism to build annotated intermediate representations. This capability is still being tested. In earlier work we developed this capability for a simple while language. The ACCEPT/W framework was a more complete implementation of ACC. However, it was impractical to demonstrate engineering significance without a broadly adopted source language.

7.1 Case Studies

The first case study we report is based on a small collection of device drivers. These include a set of NT device drivers adopted by the BLAST group and a set of Linux drivers we collected independently. For each driver we report the size of the preproccessed device driver, the size of the Boolean program calculated by ACCEPT/C, the size of the certificate calculated by BLAST, an estimate of the model validation time based on theorem proving time used by BLAST in the lazy abstraction phase, and the model checking time in Moped. Some representative results are summarized in Figure 6.

The results so far are encouraging. Model checking time is surprisingly low. It is substantially less than the estimated model validation time. While it is difficult to con-

clude anything from this amount of data, we suspect that the low model checking times has something to do with the simplicity of the domain and the safety properties studied.

	Program Size	Cert. Size	BLAST Cert. Size	Model Validation Time	Verif. Time
Linux atp.c	2482	1212	8737	$0.1s$	$0.08s$
Linux ide.c	$48K$	876	12452	$0.1s$	$0.10s$
Linux audio.c	$175K$	15410	$502K$	$0.2s$	$0.20s$
NT cdaudio.c	$456K$	$55K$	$233M$	$0.5s$	$0.15s$
NT floppy.c	$440K$	$51K$	$33M$	$0.9s$	$0.05s$

Fig. 6. Comparison between ACCEPT/C and BLAST

The second case study is also taken from the BLAST examples. It is based on a single program, an ssh server implementation. This example allows us to explore one of the initial motivations for considering the use of models as certificates: in principle one model may be able to certify multiple properties. (Ultimately we would like to validate a property that was unknown at the time the model was created, perhaps due to API evolution.)

The BLAST test bed identifies 13 separate properties to be verified.[1] The BLAST lazy abstraction algorithm discovers 17 predicates to establish these properties. We constructed Boolean programs to verify the properties individually and collectively. The results show many predicates are useful to most of the properties. Figure 7 presents a histogram that graphs the number of properties in which each predicate participated. Over half of the properties used more than eight predicates in common.

With ACCEPT/C we calculated a single Boolean Program that can be used to certify all 13 properties. This collective certificate of the 13 properties is only 25% larger than the largest certificate necessary to verify a single property. This suggests that ACC may be useful for building certificates that can support validation of multiple properties.

8 Conclusion

Abstraction-carrying code provides a framework for the certification of properties of engineering significance about programs in an untrusted environment. Experience with the ACCEPT/C toolkit shows ACC certificates are reasonably compact and that the generation, validation, and re-verification of certificates is tractable.

When compared with Temporal PCC, ACC may require more client side computation but generates significantly more compact certificates. ACC also requires a larger trust base than most PCC variants. An ACC client must trust a model checker and an automatic theorem proving tool capable of establishing the fidelity of the certificate.

[1] The BLAST test bed formulates 16 properties, but we are able to produce the results for 13 of the cases.

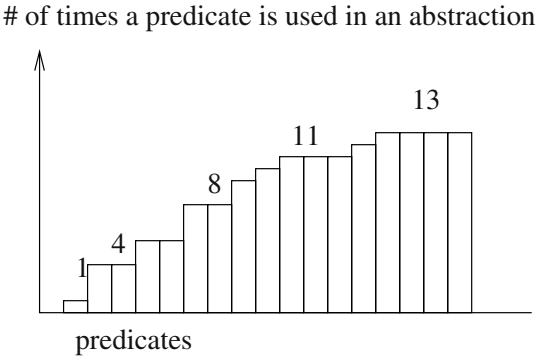


Fig. 7. The Occurrence of Predicates in the Abstractions

Future work includes the completion of the C-based ACC implementation. The most significant outstanding task is to implement client side support for the annotated intermediate language. In our preliminary investigation of ACC we have completed this task for a Java virtual machine variant. We expect the same techniques to apply.

We believe that, in general, certificates based on communicating an abstract model of a system will be more robust in use than proof based certificates. In the future a client will formulate properties based on the current versions of software found on the client system. The server will not need to have anticipated the exact configuration to provide a model that may be sufficient to validate the component. With proof based certificates such flexibility is not possible. Ultimately we expect the robustness of certificates to be a more important issue than the size of the trust base.

References

1. A. Appel. Fundational Proof-carrying Code. In *Proceeding of 16th IEEE Symposium on Logics in Computer Science*, June 2001.
2. T. Ball and S. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In *SPIN2001, Lecture Notes in Computer Science,LNCS2057*, pages 103–122. Springer-Verlag, May 2001.
3. S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In *Proceedings of CAV'98, LNCS 1427*, pages 319–331, June 1998.
4. A. Bernard and P. Lee. Temporal Logic for Proof-carrying Code. In *Proceedings of the 18th International Conference on Automated Deduction (CADE-18)*, volume 2392 of *LNAI*, pages 31–46, Copenhagen, Danmark, July 2002.
5. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Computer Aided Verification*, pages 154–169, 2000.
6. S. Das, D. Dill, and S. J. Park. Experience with Predicate Abstraction. In *Proceedings of CAV'99, LNCS 1633*, pages 160–171, Trento, Italy, July 1999.
7. S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In *Proceedings of CAV'97, Lecture Notes in Computer Science, LNCS 1254*, pages 72–83, Haifa, Israel, June 1997. Springer-Verlag.

8. T. Henzinger, R. Jhala, R. Majumdar, G. Necula, G. Sutre, and W. Weimer. Temporal-Safety Proofs for Systems Code. In *Computer-Aided Verification*, pages 526–538, 2002.
9. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages*, pages 58–70, 2002.
10. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *Proceedings of the Tenth International Workshop on Model Checking of Software (SPIN)*, pages 235–239. LNCS 2648, Springer-Verlag, 2003.
11. Kedar S. Namjoshi. Certifying Model Checkers. In *13th Conference on Computer Aided Verification (CAV)*, 2001.
12. Kedar S. Namjoshi. Lifting Temporal Proofs through Abstractions. In *VMCAI*, 2003.
13. D. Kozen. Efficient Code Certification, 1998. Technical Report, Computer Science Department, Cornell University.
14. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.
15. S. McPeak, G. C. Necula, S. P. Rahul, and W. Weimer. Cil: Intermediate languages and tools for c program analysis and transformation. In *Proceedings of Conference on Compiler Construction (CC'02)*, March 2002.
16. G. Morrisett, K. Crary, N. Glew, and D. Walker. Stacked-based Typed Assembly Language. In *Proceedings of workshop on Types in Compilation, LNCS 1473*, pages 28–52. Springer Verlag, 1998.
17. G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.
18. G. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, 1998.
19. G. Necula. A Scalable Architecture for Proof-Carrying Code, 2001.
20. Robert Kurshan. Models Whose Checks Don't Explode. In *Computer-Aided Verification*, pages 222–233, 1994.
21. H. Saidi. Model-checking Guided Abstraction and Analysis. In *Proceedings of SAS'00, LNCS 1824*, pages 377–389, Santa Barbara, CA, USA, July 2000. Springer-Verlag.
22. S. Schwoon. Moped software. available at http://wwwbrauer.informatik.tu-muenchen.de/~schwoon/moped/.
23. R. Sekar, C. Ramakrishnan, I. Ramakrishnan, and S. Smolka. Model-carrying Code(MCC): A New Paradigm for Mobile Code Security. In *New Security Paradigm Workshop*, 2001.
24. R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. DuVarney. Model-carrying code: A practical approach for safe execution of untrusted applications. In *Proceedings of ACM Symposium on Operating System Principles*, pages 15–28, 2003.
25. Z. Shao, B. Saha, V. Trifonov, and N. Papaspyrou. Type System for Certified Binaries. In *Proc. 29th ACM Symposium on Principles of Programming Languages (POPL'02)*, pages 217–232, January 2002.
26. L. Tan and R. Cleaveland. Evidence-based model checking. In *13th Conference on Computer Aided Verfication (CAV)*, pages 455–470, 2001.
27. H. Xi and R. Harper. Dependently Typed Assembly Language. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, pages 169–180, Florence, September 2001.
28. S. Xia and J. Hook. Experience with abstraction-carrying code. In *Proceedigns of Software Model Checking Workshop*, 2003.
29. S. Xia and J. Hook. An implementation of abstraction-carrying code. In *Proceedings of Foundations of Computer Security Workshop*, 2003.

Verifying Atomicity Specifications for Concurrent Object-Oriented Software Using Model-Checking*

John Hatcliff, Robby, and Matthew B. Dwyer

Department of Computing and Information Sciences, Kansas State University**

Abstract. In recent work, Flanagan and Qadeer proposed *atomicity declarations* as a light-weight mechanism for specifying non-interference properties in concurrent programming languages such as Java, and they provided a type and effect system to verify atomicity properties. While verification of atomicity specifications via a static type system has several advantages (scalability, compositional checking), we show that verification via model-checking also has several advantages (fewer unchecked annotations, greater coverage of Java idioms, stronger verification). In particular, we show that by adapting the Bogor model-checker, we naturally address several properties that are difficult to check with a static type system.

1 Introduction

Reasoning about possible interleavings of instructions in concurrent object-oriented languages such as Java is difficult, and a variety of approaches including model checking [2,4], static and dynamic [18] analyses and type systems [7] for race condition detection, and theorem-proving techniques based on Hoare-style logics [9] have been proposed for detecting errors due to unanticipated thread interleavings.

In recent work, Flanagan and Qadeer [10] noted that developers of concurrent programs often craft methods using various locking mechanisms with the goal of building method implementations that can be viewed as *atomic* in the sense that any interleaving of atomic method m's instructions should give the same result as executing m's instruction without any interleavings (i.e., in a single atomic step). They provide an annotation language for specifying the atomicity of methods, and a verification tool based on a type and effect system for verifying the correctness of a Java program annotated with atomicity specifications.

We believe that the atomicity system of Flanagan and Qadeer represents a very useful specification mechanism, in that it checks a light-weight and semantically simple property (e.g., as opposed to more expressive temporal specifications

* This work was supported in part by the U.S. Army Research Office (DAAD190110564), by DARPA/IXO's PCES program (AFRL Contract F33615-00-C-3044), by NSF (CCR-0306607) by Lockheed Martin, and by Intel Corporation.

** Manhattan KS, 66506, USA. {hatcliff,robby,dwyer}@cis.ksu.edu

B. Steffen and G. Levi (Eds.): VMCAI 2004, LNCS 2937, pp. 175–190, 2004.

(a) java.util.StringBuffer (excerpts)

```
public synchronized StringBuffer
          append(StringBuffer sb) {
  if (sb == null) { sb = NULL; }
  int len = sb.length();
  int newcount = count + len;
  if (newcount > value.length)
    expandCapacity(newcount);
  sb.getChars(0,len,value,count);
  count = newcount;
  return this;
}
```

(b) Simple dead-locking method

```
public void m1(Object p1,
               Object p2) {
  int x;
  synchronized (p1) {
    synchronized (p2) {
      x=1;
    }
  }
}
```

Fig. 1. Basic examples of non-`atomic` methods

that are harder to write) that can reveal a variety of programming errors associated with improper synchronization. For example, a common Java programming error is to assume that if a method is `synchronized`, then it is free from interference errors. However, Flanagan and Qadeer illustrate that this is not the case [10, p. 345] using the `java.util.StringBuffer` example of Figure 1(a). After `append` calls the method `sb.length` (which is also synchronized), a second thread could remove characters from `sb`. Thus, `length` becomes *stale* and no longer reflects the current length of `sb`, and so `getChars` is called with invalid arguments and throws a `StringIndexOutOfBoundsException`.

Inspired by the utility of the atomicity specifications, we investigated the extent to which these specifications could be checked effectively using model-checking. Clearly, checking via a type system can provide several advantages over model-checking: the type system approach is compositional – classes can be checked in isolation, a closing environment or test-harness need not be created, and computational complexity is lower which leads to better scalability.

Our efforts, which we report on in this paper, indicate that there are also several benefits in checking atomicity specifications using model checking when one employs a sophisticated software-dedicated model-checking engine such as Bogor [16] – an extensible software model checker that we have built as the core of the next generation of the Bandera tool set. Bogor provides state-of-the-art support for model-checking concurrent object-oriented languages including heap symmetry reductions, garbage collection, partial order reduction (POR) strategies based on static and dynamic (i.e., carried out during model-checking) escape analyses and locking disciplines, and sophisticated state compression algorithms that exploit object state sharing. Checking atomicity specifications with Bogor offers the following benefits.

- Due to the approximate and compositional nature of the type system, several forms of annotations are required for sufficient accuracy such as preconditions for each method stating which locks must be held upon entry to the method and declarations for each lock-protected object that indicates the lock that must be held when accessing that object. Much of this information can be derived automatically during Bogor's state-space exploration.
- The type system cannot handle some complex locking idioms since its static nature requires that objects and methods be statically associated with locks

via annotations. Bogor's partial order reductions strategies were developed specifically to accommodate these more complex idioms – therefore Bogor can verify many methods that rely on more complex locking schemes for atomicity.
- The type system as presented in [10] requires several forms of unchecked annotations or assumptions about objects shared between threads, and Flanagan and Qadeer note that static escape analysis could be used to partially eliminate the need for these. In our previous work on using escape analysis to enable partial order reductions [5], we concluded that the *dynamic* escape analysis as implemented in Bogor performs dramatically better than static escape analysis for reasoning about potential interference in state-space exploration. Thus, Bogor provides a very effective solution for the unchecked annotations using previously implemented mechanisms.
- The type system [10] actually fails to enforce a key condition of Lipton's reduction framework, and thus will verify as atomic some methods whose interleaving can lead to deadlock. Bogor's atomicity verification (based on the ample set partial order reduction framework which provides reductions that preserve LTL_{-X} properties as well as deadlock conditions) correctly identifies methods that violates this condition, and thus avoids classifying these interfering methods as atomic.

Finally, Bogor's aggressive state-space optimizations enable atomicity checking without significant overhead; for most of the examples reported on in [10], for example, Bogor was able to complete the checks in under one second.

We *do not* conclude that the model-checking approach is necessarily better than the type system approach to checking atomicities. As noted earlier, a type system approach to checking has several advantages over the model-checking approach.

- The type system approach, though conservative, naturally guarantees complete coverage of all behaviors of a software unit. In contrast, model-checking a software unit requires a test harness that simulates interactions that the unit might have with a larger program context. During model-checking, the behaviors that are explored are exactly those generated by the test harness. It is usually difficult to guarantee that the test harness will drive the unit through all behaviors represented by any client context, and thus some behaviors that lead to atomicity violations may be missed if the test harness is not carefully constructed. Thus, checking atomicity specifications via model-checking is typically closer to debugging than the systematic guarantees offered by the type system approach. However, it is important to note that for complex programs unchecked type annotations are often required, consequently the type system will not cover the behavior that it assumes from the annotations.
- The type system approach scales better. In addition, by its very nature, the type system approach is compositional, which allows program units to easily be checked in isolation. This enables incremental and modular checking of atomicity specifications.

We also do not conclude that *all* model checkers can effectively check atomicity specifications – using conventional model-checkers that do not provide direct support for heap-allocated data would be more difficult since information concerning locking or object-sharing is not directly represented.

Our conclusion is that type systems and model-checking are complementary approaches for checking the atomicity specification of Flanagan and Qadeer [10]. Moreover, in model-checkers such as Bogor and JPF that provide direct support for objects, checking for atomicity specifications should be included because such specifications are useful, and model-checking provides an effective verification mechanism for these specifications.

The rest of this paper is organized as follows. Section 2 provides a brief overview of Lipton's reduction theory, the atomicity type/effects system of Flanagan and Qadeer, and Bogor's enhancement of the ample set POR framework. Section 3 describes how Bogor's state-space exploration algorithm is augmented to check atomicity specifications. Section 4 presents experimental results drawn from the basic Java library examples used by Flanagan and Qadeer [10] as well as larger examples used in our previous work on partial order reductions [5]. Section 5 discusses related work, and Section 6 concludes.

2 Background

A *state transition system* [3] Σ is a quadruple (S, T, S_0, L) with a set of states S, a set of transitions T such that for each $\alpha \in T, \alpha \subseteq S \times S$, a set of initial states S_0, and a labeling function L that maps a state s to a set of primitive propositions that are true at s.

For the Java systems that we consider, each state s holds the stack frames for each thread (program counters and local variables for each stack frame), global variables (i.e., static class fields), and a representation of the heap. Intuitively, each $\alpha \in T$ represents a statement or step (e.g., execution of a bytecode) that can be taken by a particular thread t. In general, α is defined on multiple "input states", since the transition may be carried out, e.g., not only in a state s but also in another state s' that only differs from s in that it represents the result of another thread t' performing a transition on s.

For a transition $\alpha \in T$, we say that α is *enabled* in a state s if there is a state s' such that $\alpha(s, s')$ holds. Otherwise, α is disabled in s.

2.1 Lipton's Reduction Theory

Lipton introduced the theory of left/right movers to aid in proving properties about concurrent programs [14]. The motivation is that proofs can be made simpler if one is allowed to assume that a particular sequence of statements is indivisible, i.e., that the statements cannot be interleaved with statements from other threads. In order to conclude that a program P with a particular sequence of statements S is equivalent to a reduced program P/S where the statements S are executed in one indivisible transition, Lipton proposed the notion of *commuting transitions* in which particular program statements are

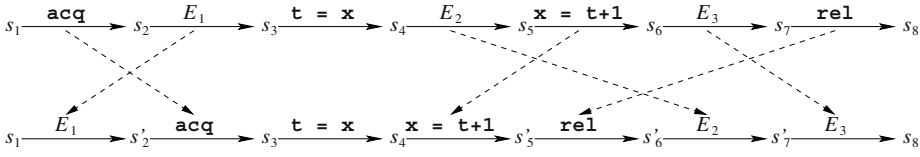


Fig. 2. Left/Right movers and atomic blocks

identified as *left movers* or *right movers*. Intuitively, a transition α is a right (left) mover if whenever it is followed (preceded) by any other transition β of a different thread, then α and β can be swapped without changing the resulting state. Concretely, Lipton established that a lock acquire (e.g., such as those at the beginning of a Java synchronized block b) is a right mover, and the lock release at the end of b is a left mover. Any read or write to a variable/field that is properly protected by a lock is both a left and right mover.

To illustrate the application of these ideas, we repeat the example given in [10]. Consider a method m that acquires a lock, reads a variable `x` protected by that lock, updates `x`, and then releases the lock. Suppose that the transitions of this method are interleaved with transitions E_1, E_2, and E_3 of other threads. Because the actions of the method m are movers (`acq` and `rel` are left and right movers, respectively, and the lock-protected assignment to `x` is both a left and right mover). Figure 2 implies that there exists an equivalent execution where the operations of m are not interleaved with operations of other threads. Thus, it is safe to reason about the method as executing in a single atomic step.

Although the original presentation of Lipton is rather informal, he does identify the property that is preserved when programs are reduced according to his theory: the set of final states of a program P equals the set of final states of the reduced program P/S; in particular P halts iff P/S halts. To achieve this property, Lipton imposes two restrictions on a set of statements S that are grouped together to form atomic statements [14, p. 719]. Restriction (R1) states that "if S is ever entered then it should be possible to eventually exit S"; Restriction (R2) states that "the effect of statements in S when together and separated must be the same." (R1) represents a fairly strong liveness requirement that can be violated if, e.g., S contributes to a deadlock or livelock, or fails to complete because it performs a Java `wait` and is never notified. (R2) is essentially stating an independence property for S: any interleavings between the statements of S do not effect the final store it produces.

2.2 Type and Effect System for Checking Atomicity

The type system of Flanagan and Qadeer assigns an atomicity a to each expression of a program where a is one of the following values: `const` (the same value is always returned each time the expression is evaluated), `mover` (the expression both left and right commutes with operations of other threads), `atomic` (the expression can be viewed as a single atomic action), and `compound` (none of the previous properties hold).

To classify a field access with a `mover` atomicity, the type system needs to know that the field is protected by a lock. The type system requires the

following program annotations to be attached to fields and methods to express this information.

- *field* `guarded_by` l: the lock represented by the lock expression l must be held whenever the field is read or written.
- *field* `write_guarded_by` l: the lock represented by the lock expression l must be held for writes, but not necessarily for reads.
- *method* `requires` $l_1, \cdots, l_n$: a method precondition that requires locks represented by lock expressions $l_1, \cdots, l_n$ to be held upon method entry; the type system verifies that these locks are held at each call-site of the method, and uses this assumption when checking the method body.

If neither guarded statement is present, the field can be read or written at any time (such accesses are classified as `atomic` since these correspond to a single JVM bytecode).

For soundness, the type system requires that each lock expression denote a fixed lock through the execution of the program. This is guaranteed in a conservative manner by ensuring that each lock expression has atomicity `const`; such expressions include references to immutable variables, accesses to final fields of `const` expressions, and calls to const methods. In particular, note that the `this` identifier of Java which refers to the receiver of the current method is `const`, and many classes in Java are synchronized by locking the receiver object.

The type system contains rules for composing statement atomicities which we do not repeat here, but we simply note that the pattern of statement atomicities required for an atomic method takes the form given by the regular expression $\texttt{mover}^*\texttt{atomic}^?\texttt{mover}^*$ (i.e., 0 or more movers followed by 0 or 1 atomic statements followed by 0 or more movers).

When considering the effectiveness of the type system for enforcing the restrictions (R1) and (R2) laid out by Lipton, there are several interesting things to note. There is no attempt by the type system to enforce the first condition (R1). This allows the method of Figure 1(b) to be assigned an `atomic` atomicity even though interleaving of its instructions can cause a deadlock when it is used in a context where a thread t_1 calls it with objects o_1, o_2 as the parameters and a thread t_2 calls it with objects o_2, o_1 as parameters. Indeed, it is difficult to imagine any type system or static analysis enforcing the condition (R1) without being very conservative (perhaps prohibitively so) or without resorting to various forms of unchecked annotations.

In many cases, the type system enforces (R2) in a very conservative way. It does not incorporate escape information which would allow accesses to fields of an object o that are not lock-protected but where o is only reachable from a single thread to be classified as `mover` rather than `atomic`. In addition, the restriction on lock expressions in annotations that only allows references from constant fields, means that more complex locking strategies are difficult to handle effectively in the type system.

2.3 Partial-Order Reductions for OO Languages

Lipton's theory for commuting transitions differs slightly from the conditions for commuting transitions used in most POR frameworks. For example, Lipton defines a transition α as a right-mover if it right commutes with *all* other

```
1  seen := {s0}
2  pushStack(s0)
3  DFS(s0)

DFS(s)
4  workSet := ample(s)
5  while workSet ≠ ∅ do
6      let α ∈ workSet
7      workSet := workSet \ {α}
8      s' := α(s)
9      if s' ∉ seen then
10         seen := seen ∪ {s'}
11         pushStack(s')
12         DFS(s')
13         popStack()
end DFS
```

Fig. 3. Depth-first search with partial order reduction (DFS-POR)

transitions β from other threads. Commutativity properties in POR frameworks are often captured by a symmetric independence relation I on transitions such that $I(\alpha, \beta)$ implies **(a)** if $\alpha, \beta \in \mathit{enabled}(s)$ then $\alpha \in \mathit{enabled}(\beta(s))$, and **(b)** if $\alpha, \beta \in \mathit{enabled}(s)$ then $\alpha(\beta(s)) = \beta(\alpha(s))$. The two notions of commutativity differ in the following ways. A right mover must right-commute with *all* transitions from other threads, while the independence relation allows a transition to commute with only *some* other transitions. In addition, the commutativity definitions in the Lipton theory are asymmetric whereas, the independence relation I is required to be symmetric. This fact, along with the condition (a) on I implies that a transition α that is a lock acquire cannot be independent of another transition β that acquires the same lock because α can disable β by acquiring the lock. However, the conditions of Lipton allow an acquire to be a right mover (see [14, p. 719] for details).

In recent work [5], we described a partial order reduction framework for model-checking concurrent object-oriented languages that we have implemented in Bogor. This framework is based on the *ample set* approach of Peled [15] and detects independent transitions using locking information as inspired by the work of Stoller [19], reachability properties of the heap, and dynamic escape analysis. Independent transitions detected in our framework include accesses to fields that are always lock-protected or read-only, accesses to fields of objects that are only reachable through lock-protected objects, and accesses to *thread-local* objects, i.e., objects that are only reachable from a single thread at a given state s, etc. Because Bogor maintains an explicit representation of the heap, it is quite easy and efficient to check the heap properties necessary to detect the independence situations listed above. Independence properties for thread-local (non-escaping) objects are detected automatically by scanning the heap. Independence associated for read-only fields require a `read-only` annotation on a field, and a simple `lock-protected` annotation on a field that is protected by a lock (both of these annotations are checked for violations during model-checking). Note that our locking annotation is simpler than the annotation of Flanagan and Qadeer in that one does not need to give a static expression indicating the guarding lock. Dynamically scanning the heap during model checking allows independence to be detected in a variety of settings, e.g., when the lock protecting an object changes during program execution or that locking is not required in states where the object associated with the field is thread-local.

Figure 3 presents the depth-first search state-exploration algorithm that incorporates the ample set partial order reduction strategy [3, p. 143]. Space constraints do not allow a detailed explanation, but the intuition of the approach

is a follows. At each state s with outgoing enabled transitions $\mathit{enabled}(s) = \{\alpha_1, \ldots \alpha_n, \beta_1, \ldots \beta_m\}$, $\mathit{ample}(s)$ returns a subset $\{\alpha_1, \ldots \alpha_n\}$ to expand. The selection of $\mathit{ample}(s)$ is constrained in such a way as to guarantee that the reduced system explored by Figure 3 satisfies the same set of LTL$_{-X}$ formulas as the unreduced system (in which all enabled transitions from each state are expanded). Following [3, p. 158], our strategy for choosing $\mathit{ample}(s)$ is to find a thread t such that all of its transitions at s are independent of transitions from all other threads at s, and return that set $\mathit{enabled}(s,t)$ as the value of $\mathit{ample}(s)$ (technically, a few other conditions must be satisfied to preserve LTL$_{-X}$ properties). If such a thread t cannot be found, $\mathit{ample}(s)$ is set to $\mathit{enabled}(s)$. We refer the reader to [5] for details.

3 Model-Checking Atomicity Specifications

We have presented two frameworks for describing commuting transitions: the Lipton mover framework, and the independence framework used in POR. We now describe how atomicity specifications can be checked by augmenting Figure 3 to classify transitions according to each of these frameworks. In effect, this yields two approaches for checking atomicity specifications using model checking: MC-mover which will classify transitions according to Lipton's framework and MC-ind which classifies transitions according to the independence framework. During our presentation, we will compare these approaches and contrast them with the Type-System checking approach of Flanagan and Qadeer.

In our atomicity checker, a method m or block of code can be annotated as `compound` (the default specification which every method trivially satisfies), `atomic`, or `mover`. Intuitively, a `mover` method is a special case of an `atomic` where all transitions are both left and right movers according to MC-mover or all independent according to MC-ind. For simplicity, we will only describe checking `atomic` methods since it is obvious that checking the patterns for `mover` will only require very minor changes.

In MC-ind, transitions can be classified uniquely as independent (I) or dependent (D) in Bogor by leveraging locking and heap structure information. In MC-mover, transitions can be classified as left-mover (L), right-mover (R), or atomic (A), and we allow a single transition to have both L and R classifications.

For an `atomic` method m, the goal of MC-ind checking is to guarantee that the transitions of m follow the pattern $\mathsf{I}^*\mathsf{D}^?\mathsf{I}^*$ every time method m is encountered during the state-space search. For each thread t, the checker uses an internal *region position* variable held in the state-vector with values (N,I,D) to mark where t's program counter (pc) is positioned with respect to the required transition pattern. N indicates t's pc is not in a block annotated as atomic, I indicates t's pc is in an atomic block, but has not yet executed a D transition, and D indicates t's pc is in an atomic block, and has already executed a D transition. Checking using MC-mover looks for the pattern $\mathsf{R}^*\mathsf{A}^?\mathsf{L}^*$ with region position values (N,R,L) where R indicates that t's pc is in an atomic block but has not yet executed an A or L transition, and L indicates t has executed an A (i.e., t's pc is moving into or through the L region of the pattern).

```
...
 2.1  M_ind := []
 2.2  R^⊗ := ∅
 2.3  R^×_ind := ∅
...
 3.1  foreach ρ^⊗ ∈ R^⊗ do
 3.2      signal ρ^⊗ is definitely not atomic
 3.3  foreach ρ^× ∈ R^×_ind do
 3.4      signal ρ^× is possibly not atomic
...
 4.1  checkAtomicDeadlock(s, workSet)
...
 6.1      t := thread(α)
 6.2      σ_ind := getRegionPosition(t, M_ind)
...
 8.1      updateIndAtomic(s, s', α)
...
13.1      else checkAtomicCycle(s', t)
13.2      M_ind := M_ind[t ↦ σ_ind]
...

checkAtomicCycle(s, t)
36  if isInStack(s) then
37      if region_atom?(s, t) ≠ ∅ then
38          R^⊗ := R^⊗ ∪ region_atom?(s, t)
end checkAtomicCycle

updateIndAtomic(s, s', α)
39  t := thread(α)
40  σ := getRegionPosition(t, M_ind)
41  if region_atom?(s', t) = ∅ then
42      if σ ≠ N then
43          M_ind := M_ind[t ↦ N]
44  elseif isWait(α) then
45      R^⊗ := R^⊗ ∪ region_atom?(s', t)
46  elseif σ = N then
47      M_ind := M_ind[t ↦ I]
48  elseif σ = I ∧ ¬isIndependent(s, α) then
49      M_ind := M_ind[t ↦ D]
50  elseif σ = D ∧ ¬isIndependent(s, α) then
51      R^×_ind := R^×_ind ∪ region_atom?(s', t)
end updateIndAtomic

checkAtomicDeadlock(s, workSet)
52  if workSet = ∅ then
53      foreach t ∈ threads(s) do
54          if region_atom?(s, t) ≠ ∅ then
55              R^⊗ := R^⊗ ∪ region_atom?(s, t)
56  elseif ∃θ ⊆ threads(s).
57                  circSync(θ) then
58      if ∃t ∈ θ.region_atom?(s, t) ≠ ∅ then
59          R^⊗ := R^⊗ ∪ region_atom?(s, t)
end checkAtomicDeadlock
```

Fig. 4. Additions to DFS-POR for MC-ind

In addition to checking that m conforms to the required pattern of transitions, the checker will also flag as "unverified" any `atomic` or `mover` method where a thread that cannot eventually move past the end of that method. We have two approaches for implementing this. The first uses the standard approach of Spin [12] to detect invalid end-states (completions of execution where a thread has not moved past the end of an atomic method) as well as nested depth-first search to look for non-progress cycles (indicating live-locks). Nested depth-first search makes this approach more expensive, so we implemented a cheaper scheme that looks directly for cycles in a lock-holding graph and can catch many common cases.

Figure 4 presents the MC-ind algorithm. Due to space constraints, we have elided the parts of the algorithm (using ...) that do not change from the basic DFS algorithm presented in Figure 3. Line numbers $x.y$ of Figure 4 are inserted after the line number x of Figure 3.

The input of the algorithms is a set of regions $\mathcal{R}_{atom?}$ where each region represents a method or block of code that is annotated as `atomic`. In this presentation, we represent a region $\rho \in \mathcal{R}_{atom?}$ as a non-empty finite sequence of transitions $[\alpha_1, ..., \alpha_n]$ drawn from the same thread. For simplicity, we assume that regions do not overlap with each other, and there are no loop edges across regions.[1] Given a thread t and a state s, we define the function $region_{atom?}$ to return a singleton set containing the region $\rho \in \mathcal{R}_{atom?}$ if one of t's transition at state s is in the sequence of ρ. Otherwise, it returns the empty set.

[1] Our actual implementation handles all of Java, and allows e.g., atomic methods to call other atomic methods, etc.

```
...
 2.1  M_mov := []
...
 2.3  R^×_mov := ∅
...
 3.3  foreach ρ^× ∈ R^×_mov do
...
 6.2      σ_mov := getRegionPosition(t, M_mov)
...
 8.1      updateMovAtomic(s, s', α)
...
13.2      M_mov := M_mov[t ↦ σ_mov]
...
```

```
updateMovAtomic(s, s', α)
60  t := thread(α)
61  σ := getRegionPosition(t, M_mov)
62  if region_atom?(s', t) = ∅ then
63      if σ ≠ N then
64          M_mov := M_mov[t ↦ N]
65  elseif isWait(α) then
66      R^⊗ := R^⊗ ∪ region_atom?(s', t)
67  elseif σ = N then
68      M_mov := M_mov[t ↦ R]
69  elseif σ = R ∧ ¬isRightMover(α) then
70      M_mov := M_mov[t ↦ L]
71  elseif σ = L ∧ ¬isLeftMover(α) then
72      R^×_mov := R^×_mov ∪ region_atom?(s', t)
end updateMovAtomic
```

Fig. 5. Additions to DFS-POR for MC-mover

Region positions are maintained using the position table $\mathcal{M}_{ind}$ (line 2.1) that maps each thread t to its position status $\{N, I, D\}$. Given a thread t and a position table $\mathcal{M}$, we define the function *getRegionPosition* to return N if $t \notin dom(\mathcal{M})$; otherwise, it returns $\mathcal{M}(t)$.

Note that transition classifications are necessarily conservative since the analysis must be safe, and so a method may actually be atomic, yet fail to be classified as such because of imprecision in the Bogor independence detection algorithm. Yet there are some cases where we can tell that there is definitely an atomicity violation (e.g., if a transition executes a wait lock operation, or a deadlock or cycle within the region is found). We wish to distinguish these cases when reporting to the user, and we use the set $\mathcal{R}^{\otimes} \subseteq \mathcal{R}_{atom?}$ to hold the regions that *definitely* violate their atomicity annotations (line 2.1), so we use the set $\mathcal{R}^{\times}_{ind} \subseteq \mathcal{R}_{atom?}$ to hold the regions that *possibly* violate their atomicity annotations as determined by transition classifications (lines 2.2-2.3).

The algorithm proceeds as follows (for liveness issues, we include an explanation of our less expensive checks, since the strategy using the nested depth-first search is well-documented [12]). Once the ample set is constructed for the current state (line 4), we check if there exists a thread such that it is in one of its atomic regions and it is in a (fully/partially) deadlock state (line 4.1 and lines 52-59). If a deadlock exists, then the atomic region cannot really be atomic (given a set of threads θ, we define the function *circSync* to return *True* if there is cycle in the lock-holding/acquiring relation for threads of θ). Once a transition α is selected from the work set, then we save the region position of the thread t that is executing α (lines 6.1-6.2) so that we can restore the position marker when backtracking later on (line 13.2). After α is executed, we update the region position of t (line 8.1). If the next state s' has been seen before, we check if s' is in the stack. If it is, then there is a cycle (non-terminating loop) in the state space. Thus, if t is in the same atomic region at state s', then that region definitely cannot be atomic (line 13.1 and lines 36-38).

The region position is updated as follows (*updateIndAtomic*). If t is not in one of its atomic regions at state s', and if t's status at state s was other than N, then the status is updated to N (lines 41-43). If α is a wait transition, then the annotated atomic region where t is at state s' definitely cannot be atomic (line

(a) "Corrected" StringBuffer (excerpts)

```
public synchronized StringBuffer
          append(StringBuffer sb) {
  synchronized (sb) {
    if (sb == null) { sb = NULL; }
    int len = sb.length();
    int newcount = count + len;
    if (newcount > value.length)
      expandCapacity(newcount);
    sb.getChars(0,len,value,count);
    count = newcount;
    return this;
  }
}
```

(b) Replicated Workers (excerpts)

```
public final synchronized
                  Vector take() {
  Vector returnVal =
    new Vector(theCollection
                .size());
  for (int i = 0;
       i < theCollection.size();
       i++) {
    returnVal.addElement
      (theCollection.take());
  }
  return returnVal;
}
```

Fig. 6. Java examples illustrating atomicity checking by invisibility vs. Lipton's movers

44-45). If t's status was N, then we can infer that t has just entered one of its atomic regions. Thus, the status is updated to I (lines 46-47). If t's status was I and α is not an independent transition at state s, then t's status is updated to D (lines 48-49). Lastly, if t's status was D and α is not an independent transition, then we can infer that we have seen two non-independent transitions inside the atomic region. Thus, the required atomicity pattern has been violated (lines 50-51). The checking algorithm for MC-mover is similar and is presented in Figure 5.

Note that considering both styles of transition classification with MC-ind and MC-mover rather than some hybrid approach allows us to highlight the subtle differences between the frameworks. Also, using the approaches as they currently exist allows us to appeal to previous correctness results for both frameworks instead of proving the correctness of a combined approach. Taking MC-ind as an example, if the algorithm confirms that a method m is atomic with respect to a given test harness (i.e., environment that closes the system and provides the calls into the classes under analysis), then, for that particular test harness, we can conclude that m always completes, and previously established correctness results for our notion of independence [5] allow us to conclude that there is no interference with the statements of the methods.

3.1 Assessing MC-ind vs. MC-mover

Recall the `StringBuffer` example of Figure 1(a) which was not atomic because threads could interfere with the variable `sb` between a sequence of reads. Figure 6(a) attempts to solve this problem by locking the `sb` variable. With this modification, Type-System would verify that this method is atomic. However, in a manner similar to Figure 1(b), this method can cause a deadlock if `append` is called from two different threads with the receiver object and method parameter reversed in the calls. Using a test harness with two threads that call append with arguments in opposing orders, both MC-ind and MC-mover would detect that the method is non-atomic.

```
public abstract class RWVSN {
  protected Vector waitingWriterMonitors_ = new Vector();
  ...
  protected synchronized void notifyWriter() {
    if (waitingWriterMonitors_.size() > 0) {
      Object oldest = waitingWriterMonitors_.firstElement();
      waitingWriterMonitors_.removeElementAt(0);
      synchronized(oldest) { oldest.notify(); }
      ++activeWriters_;
  }}}
```

Fig. 7. A readers/writers example (excerpts)

Now consider Figure 6(b), which is taken from the replicated worker framework presented in [6]. In this method, there is a series of lock acquires/releases on the vectors referenced by `theCollection` and `returnVal` when the `size()` and `take()` methods are called. Thus, using the mover transition classification, the transitions have the form ..R..L..R..L which means that neither Type-System nor MC-mover can confirm that this method is atomic. However, due to the way that the data structures are set up, the vector referenced by the `theCollection` field is always dominated by the receiver object's lock (and thus protected by it). In addition, our dynamic escape analysis can detect that the `returnVal` local variable refers to an object that is thread-local at every point it is accessed during the method. Using our independence classification, the transitions (even the acquire/releases) have the form I^+, and therefore MC-ind can verify that the method is indeed atomic.

Thus, there are some cases where MC-ind can detect atomicity but MC-mover cannot (in the case above, due to the fact that the dynamic escape analysis detects that the lock acquires are not actually needed to obtain non-interference *for that method*). Similarly, there are cases when MC-mover can detect atomicity but MC-ind cannot. This typically occurs when there is a method with nested lock acquire/release which has a pattern RR..LL in MC-mover but DD..II in MC-ind. Since both methods are conservative but safe with respect to the supplied test harness, even if only one method concludes that a method is `atomic`, then we can safely conclude that the method is atomic with respect to the supplied test harness. We take advantage of this in an alternate implementation that runs both methods simultaneously. Specifically, we weave line 2.1, 2.3, 4.3, 8.1, and 13.2 of Figure 4 and Figure 5 and replace line 3.3 as follows:

$$3.3\ \textbf{foreach}\ \rho^{\times} \in \mathcal{R}^{\times}_{ind} \cap \mathcal{R}^{\times}_{mov}\ \textbf{do}$$

That is, only if both algorithms agree that a region is possibly not atomic do we report that it is *possibly* not atomic. As will be discussed in the experiment section, the combination of the algorithms is precise enough to determine the methods in Figure 6 are atomic.

The approach can be further improved by noting that methods that fail MC-mover due to patterns like ..acq l_1..rel l_1..acq l_2..rel l_2.. (i.e., R..L..R..L) often are still atomic because l_1 and l_2 are from objects o_1 and o_2 that are either thread-local or already dominated by (and thus protected by) another lock. Figure 7 presents a readers/writers example[2] that illustrates this case (note

[2] `http://gee.cs.oswego.edu/dl/cpj/classes/RWVSN.java`

Table 1. Experiment Data

Example			**(I)**	**(L)**	**(C)**	**(W)**
Original String-buffer	Trans: 383	**check**	9	9	9	9
Threads: 3	States 9	**noise**	0	0	0	0
Locations: 175	Time: .26	**error**	1	1	1	1
Deadlock String-buffer	Trans: 99	**check**	9	9	9	9
Threads: 3	States: 4	**noise**	0	0	0	0
Locations: 183	Time: .13	**error**	1	1	1	1
Readers Writers	Trans: 127337	**check**	23	23	23	26
Threads: 5	States: 2504	**noise**	3	3	3	0
Locations: 314	Time: 19	**error**	0	0	0	0
Replicated Workers	Trans: 230894	**check**	38	38	39	39
Threads: 4	States: 4477	**noise**	1	1	0	0
Locations: 509	Time: 63.88	**error**	0	0	0	0

the sequence of acquire/release pairs represented by the method calls on the `waitingWriterMonitors_` and the synchronization on `oldest`). However, the read-only `waitingWriterMonitors_` field points to a `Vector` object that is lock dominated (hence protected) by the receiver `RWVSN` object.

Based on this observation, we can modify the existing mover classification (in which an acquire l is never considered a left mover except when it actually represents a re-acquire of l) to take into account thread-locality and lock domination (protection) information accumulated by Bogor:

- In the right mover region of an atomic region, a left mover is now defined to be a lock release when the lock being released is non-thread-local or not protected by some locks held by the current executing thread.
- In the left mover region of an atomic region, a right mover is now defined to be a lock acquire when the lock being acquired is non-thread-local or not protected by some locks held by the current executing thread.

That is, we avoid changing the region position information when encountering lock acquires/releases that do not actually play a role in protecting an object. This change allows MC-mover to correctly identify the method of Figure 7 as atomic since the lock operations on `waitingWriterMonitors_` do not change the the region position (i.e., one has the pattern R^+ up to the point where the lock on `object` is released, and the remaining increment can be classified as L).

Note that Type-System cannot establish that this method is atomic because its rule for `synchronize` l does not recognize the fact that some other lock may already by protecting l, nor does it take into account thread-locality information.

4 Experimental Results

Figure 1 presents the results of running the examples that we have described previously on an Opteron 1.8 GHz (32-bit mode) with maximum heap of 1 Gb using the Java 2 Platform (we have also run almost all of the examples from [10] with results similar to what is shown above). In all runs, we used all reductions available in Bogor described in [5] with the addition of the read-only reduction [19] The **(I)**, **(L)**, **(C)**, and **(W)** denote the MC-ind, MC-mover, the combination of MC-ind and MC-mover and and the combination that uses the modified definition of left/right movers presented at the end of the previous section. For each example, we give the number of threads and the locations or

control points of the example that are executed. Furthermore, we give the number of transitions and states, and time (in seconds) needed to run each test case. The maximum memory consumption of the examples is 2.69 Mb memory, which is for the replicated workers example, and the minimum memory consumption is .8 Mb for the String-buffer example. For each example, **check** is the number of atomic methods that are verified, **noise** is the number of atomic methods that cannot be verified due to imprecision of the algorithm, and **error** is the number of specified atomic methods that definitely cannot be atomic.

For Original String-buffer, we correctly detect the atomicity violation of `append` as reported by [10], and for Deadlock String-buffer, we correctly detect the atomicity violation due to deadlock that is not detected by [10]. Note, however that this relied on constructing a test harness that happened to expose the deadlocking schedule. Note that the Readers/Writers and the Replicated Workers example contains Java synchronized collection library such as the `java.lang.Vector` class that are also indirectly verified for atomicity via calls from the larger examples. In addition, we verify the `notifyReaders`, `notifyWriter`, `afterRead`, and `afterWrite` methods of the Readers Writers example as atomic as well as various methods for synchronization in the Replicated Workers.

The timing numbers indicate the model-checking approach is feasible for unit-testing. Fewer annotations are required – we do not require the pre-condition annotations of Type-System that indicate the locks that are held when a method is called, and although we do require annotations stating that fields are lock protected, our approach infers the protecting objects instead of having them stated directly. Flanagan and Qadeer note that they run all their experiments with an unchecked annotation that allows their checker to assume that an object does not escape its constructor. Such a property is automatically checked in Bogor's dynamic escape analysis.

We have already noted that to apply our model-checking approach, the user must construct a test harness (environment) that generates appropriate coverage of the system's execution paths. It is possible that the model-checking approach can fail to detect errors because the behavior of the environment is not sufficiently rich. Also note that the model-checking approaches will likely check a method m many times during the course of verification whereas the type system only needs to make one pass. Of course the benefit is an increase in the precision due to customization of reasoning to the invocation contexts.

5 Related Work

We have already made extensive comparisons with Flanagan and Qadeer's type system [10], which inspired the work presented in this paper. Their type system is based on the earlier type system for detecting race conditions [7]. Flanagan and Freund developed Atomizer [8], a runtime verification tool to check atomicity specifications that uses a variant of the Eraser algorithm to detect the lock-sets that protect shared variables and thread-local variables similar to [5]. However, in their algorithm, shared variables cannot become unshared later on, and they also failed to enforce Lipton's R1 condition. Wang and Stoller [21] also

developed a similar runtime tool for checking atomicity. One notable feature of their tool is that given a trace, the tool permutes the ordering of events to find atomicity violations. Both of these run-time checking approaches scale better than our model-checking approach because they do not store states and they do not consider every possible interleaving. Of course, this means that they are also more likely to fail to detect atomicity violations since many feasible paths are not examined.

Our previous work on partial order reductions [5] combined escape analysis with lock-based reduction strategies formalized earlier by Stoller [19] and implemented in JPF [2]. Stoller and Cohen have recently developed a theory of reductions using abstract algebra [20], and their framework addresses many of the issues related to independence and commutativity discussed here.

Other interesting efforts on software model-checking such as [1,11] have not considered many of the issues addressed here, since those projects have focused on automated abstraction of sequential C programs and have not included concurrency nor dynamically created objects.

Finally, model-checkers such as Spin [13] include the keyword `atomic`. However, this represents a directive to the model-checker to group transitions together without any interleaving rather than a specification to be verified against an implementation in which a developer has tried to achieve non-interference using synchronization mechanisms.

6 Conclusion

Flanagan and Qadeer have argued convincingly that atomicity specifications and associated verification mechanisms are useful in the context of concurrent programming. We believe that our work demonstrates that model-checking using state-of-the-art software model-checkers like Bogor provides an effective means of checking atomicity specifications. The key enabling factors are (1) Bogor's partial order reduction strategies based on dynamically accumulated locking and object escape information that is more difficult to obtain in static type systems, and (2) model-checking enables the verification process to easily enforce the "must complete" requirement from Lipton's theory which was not enforced in the type system of Flanagan and Qadeer (and indeed, would be difficult to enforce in any type system without resorting to very conservative approximations). An extended version of this paper and more information about examples and experimental results can be found on the Bogor web site [17].

References

1. T. Ball and S. Rajamani. Bebop: a symbolic model-checker for boolean programs. SPIN 2000, LNCS 1885, pages 113–130. Springer-Verlag, 2000.
2. G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder – a second generation of a Java model-checker. In *Proceedings of the Workshop on Advances in Verification*, July 2000.
3. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

4. J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera : Extracting finite-state models from Java source code. ICSE, 2000.
5. M. B. Dwyer, J. Hatcliff, V. Ranganath, and Robby. Exploiting object escape and locking information in partial order reduction for concurrent object-oriented programs. Technical Report TR2003-1, SAnToS Laboratory, Kansas State University, 2003.
6. M. B. Dwyer and V. Wallentine. A framework for parallel adaptive grid simulations. *Concurrency : Practice and Experience*, 9(11):1293–1310, Nov. 1997.
7. C. Flanagan and S. N. Freund. Type-based race detection for Java. PLDI, 2000.
8. C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *Proceedings of POPL*, 2003. (to appear)
9. C. Flanagan, S. N. Freund, and S. Qadeer. Thread-modular verification for shared-memory programs. ESOP, 2000.
10. C. Flanagan and S. Qadeer. A type and effect system for atomicity. PLDI, 2003.
11. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with Blast. In T. Ball and S. Rajamani, editors, *Proceedings of 10th International SPIN Workshop*, LNCS 2648, pages 235–239. Springer-Verlag, 2003.
12. G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall International, 1991.
13. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.
14. R. J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12), 1975.
15. D. Peled. Combining partial order reductions with on-the-fly model-checking. In D. Dill, editor, *Proceedings of the 1994 Workshop on Computer-Aided Verification (LNCS 818)*, pages 377–390. Springer, 1994.
16. Robby, M. B. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular model checking framework. In *Proceedings of the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2003.
17. Bogor Website. |http://bogor.projects.cis.ksu.edu—, 2003.
18. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.
19. S. Stoller. Model-checking multi-threaded distributed Java programs. In *International Journal on Software Tools for Technology Transfer*. Springer-Verlag, 2002.
20. S. Stoller and E. Cohen. Optimistic synchronization-based state-space reduction. In H. Garavel and J. Hatcliff, editors, *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (LNCS 2619)*, 2003.
21. L. Wang and S. D. Stoller. Run-time analysis for atomicity. In *Proceedings of the Workshop on Runtime Verification*, volume 89.2 of ENTCS, 2003.

Static Analysis versus Software Model Checking for Bug Finding

Dawson Engler and Madanlal Musuvathi

Computer Systems Laboratory
Stanford University
Stanford, CA 94305, U.S.A.
{engler,madan}@cs.stanford.edu

1 Introduction

This paper describes experiences with software model checking after several years of using static analysis to find errors. We initially thought that the trade-off between the two was clear: static analysis was easy but would mainly find shallow bugs, while model checking would require more work but would be strictly better — it would find more errors, the errors would be deeper, and the approach would be more powerful. These expectations were often wrong.

This paper documents some of the lessons learned over the course of using software model checking for three years and three projects. The first two projects used both static analysis and model checking, while the third used only model checking, but sharply re-enforced the trade-offs we had previously observed.

The first project, described in Section 3, checked FLASH cache coherence protocol implementation code [1]. We first used static analysis to find violations of FLASH-specific rules (e.g., that messages are sent in such a way as to prevent deadlock) [2] and then, in follow-on work, applied model checking [3]. A startling result (for us) was that despite model checking's power, it found far fewer errors than relatively shallow static analysis: eight bugs versus 34.

The second project, described in Section 4, checked three AODV network protocol [4] implementations. Here we first checked them with CMC [5], a model checker that directly checks C implementations. We then statically analyzed them. Model checking worked well, finding 42 errors (roughly 1 per 300 lines of code), about half of which involve protocol properties difficult to check statically. However, in the class of properties both methods could handle, static analysis found more errors than model checking. Also, it took much less effort: a couple of hours, while our model checking effort took approximately three weeks.

The final project, described in Section 5, used CMC on the Linux TCP network stack implementation. The most startling result here was just how difficult it is to model check real code that was not designed for it. It turned out to be easier to run the *entire* Linux Kernel along with the TCP implementation in CMC rather than cut TCP out of Linux and make a working test harness. We found 4 bugs in the Linux TCP implementation.

The main goal of this paper is to compare the merits of the two approaches for finding bugs in system software. In the properties that could be checked

B. Steffen and G. Levi (Eds.): VMCAI 2004, LNCS 2937, pp. 191–210, 2004.

by both methods, static analysis is clearly more successful: it took less time to do the analysis and found more errors. The static analysis simply requires that the code be compiled, while model checking a system requires a carefully crafted environment model. Also, static analysis can cover *all* paths in the code in a straightforward manner. On the other hand, a model checker executes only those paths that are explicitly triggered by the environment model. A common misconception is that model checking does not suffer from false errors, while these errors typically inundate a static analysis result. In our experience, we found this not to be true. False execution paths in the model checker can be triggered by erroneous environments, leading to false errors. These errors can be difficult to trace and debug. Meanwhile, false errors in static analysis typically arise out of infeasible paths, which can be eliminated by simple analysis or even unsubstantial manual inspection.

The advantage of model checking is in its ability to check for a richer set of properties. Properties that require reasoning about the system execution are not amenable to static checking. Many protocol specific properties such as routing loops and protocol deadlocks fall in this category. A model checker excels in exploring intricate behaviors of the system and finding errors in corner cases that have not been accounted for by the designers and the implementors of the system. However, the importance of checking these properties should significantly over-weigh the additional effort required to model check a system.

While this paper describes drawbacks of software model checking compared to static analysis, it should not be taken as a jeremiad against the approach. We are very much in the "model checking camp" and intend to continue research in the area. One of the goals of this paper is to recount what surprised us when applying model checking to large real code bases. While more seasoned minds might not have made the same misjudgments, our discussions with other researchers have shown that our naivete was not entirely unreasonable.

2 The Methodologies

This paper is a set of case studies, rather than a broad study of static analysis and model checking. While this limits the universality of our conclusions, we believe the general trends we observe will hold, though the actual coefficients observed in practice will differ.

2.1 The Model Checking Approach

All of our case studies use traditional explicit state model checkers [6, 7]. We do no innovation in terms of the actual model checking engine, and so the challenges we face should roughly mirror those faced by others. We do believe our conclusions optimistically estimates the effort needed to model check code. A major drawback of most current model checking approaches is the need to manually write a specification of the checked system. Both of our approaches dispense with

this step. The first automatically extracts a slice of functionality that is translated to the model checking language, similar to the automatic extraction work done by prior work, notably Bandera [8] and Feaver [9]. Our second approach eliminates extraction entirely by directly model checking the implementation code. It is similar to Verisoft [10], which executes C programs and has been successfully used to check communication protocols [11] and Java PathFinder [12], which uses a modified Java virtual machine that can check concurrent Java programs.

2.2 The Static Analysis Approach

The general area of using static analysis for bug finding has become extremely active. Some of the more well-known static tools include include PREfix [13], ESP [14], ESC [15], MOPS [16] and SLAM [17], which combines aspects of both static analysis and model checking.

The static tool approach discussed in this paper is based on compiler extensions ("checkers") that are dynamically linked into the compiler and applied flow-sensitively down a control-flow graph representation of source code [18]. Extensions can perform either intra- or inter-procedural analysis at the discretion of the checker writer. In practice, this approach has been effective, finding hundreds to thousands of errors in Linux, BSD, and various commercial systems.

While we make claims about "static analysis" in general, this paper focuses on our own static analysis approach ("metacompilation" or MC), since it is the one we have personal experience. The approach has several idiosyncratic features compared to other static approaches that should be kept in mind. In particular, these features generally reduce the work needed to find bugs as compared to other static analysis techniques.

First, our approach is unsound. Code with errors can pass silently through a checker. Our goal has been to find the maximum number of bugs with the minimum number of false positives. In particular, when checkers cannot determine a needed fact they do not emit a warning. In contrast, a sound approach must conservatively emit error reports whenever it cannot prove an error cannot occur. Thus, unsoundness lets us check effectively properties that done soundly would overwhelm the user with false positives.

Second, we use relatively shallow analysis as compared to a simulation-based approach such as in PREfix [13]. [1] Except for a mild amount of path-sensitive analysis to prune infeasible paths [19], we do not: model the heap, track most variable values, or do sophisticated alias analysis. A heavier reliance on simulation would increase the work of using the tool, since these often require having to build accurate, working models of the environment and of missing code. In a sense simulation pushes static analysis closer to model checking, and hence shares some of its weaknesses as well as strengths.

[1] Our approach has found errors in code checked by PREfix for the same properties, so the depth of checking is not entirely one-sided.

Third, our approach tries to avoid the need for annotations, in part by using statistical analysis to infer checkable properties [20]. The need for annotations would dramatically increase the effort necessary to use the tool.

3 Case Study: FLASH

This section describes our experience checking FLASH cache coherence protocol code using both static analysis and model checking [1]. The FLASH multiprocessor implements cache coherence in software. While this gives flexibility it places a serious burden on the programmer. The code runs on each cache miss, so it must be egregiously optimized. At the same time a single bug in the controller can deadlock or livelock the entire machine.

We checked five FLASH protocols with static analysis and four with model checking. Protocols ranged from 10K to 18K lines and have long control flow paths. The average path was 73 to 183 lines of code, with a maximum of roughly 400 lines. Intra-procedural paths that span 10-20 conditionals are not uncommon. For our purposes, this code is representative of the low-level code used on a variety of embedded systems: highly optimized, difficult to read, and difficult to get correct. For the purpose of finding errors, FLASH was a hard test: by the time we checked it had already undergone over five years of testing under simulation, on a real machine, and one protocol had even been model checked using a manually constructed model [21].

3.1 Checking FLASH with Static Analysis

While FLASH code was difficult to reason about, it had the nice property that many of the rules it had to obey mapped clearly to source code and thus were readily checked with static analysis. The following rule is a representative example. In the FLASH code, incoming message buffers are read using the macro `MISCBUS_READ_DB`. All reads must be preceded by a call to the macro `WAIT_FOR_DB_FULL` to synchronize the buffer contents. To increase parallelism, `WAIT_FOR_DB_FULL` is only called along paths that require access to the buffer contents, and it is called as late as possible along these paths. This rule can be checked statically by traversing all program paths until we either (1) hit a call to `WAIT_FOR_DB_FULL` (at which point we stop following that path) or (2) hit a call to `MISCBUS_READ_DB` (at which point we emit an error). In general the static checkers roughly follow a similar pattern: they match on specific source constructs and use a extensible state machine framework to ensure that the matched constructs occur (or do not occur) in specific orders or contexts.

Table 1 gives a representative listing of the FLASH rules we checked. Since the primary job of a FLASH node is to receive and respond to cache requests, most rules involve correct message handling. The most common errors were not deallocating message buffers (9 errors) and mis-specifying the length of a message (18 errors). The other rules were not easier, but generally had less locations where they had to be obeyed. There were 33 errors in total and 28 false positives. We

Table 1. Representative FLASH rules the number of lines of code for a MC rule checker (**LOC**), the number of bugs the checker found (**Bugs**) as well as the number of false positives (**FP**). We have elided other less useful checkers; in total, they found one more bug at a cost of about 30 false positives.

Rule	LOC	Bugs	False
`WAIT_FOR_DB_FULL` must come before `MISCBUS_READ_DB`	12	4	1
The `has_data` parameter for message sends must match the specified message length (be one of `LEN_NODATA`, `LEN_WORD`, or `LEN_CACHELINE`).	29	18	2
Message buffers must be: allocated before use, deallocated after, and not used after deallocation.	94	9	25
Message handlers can only send on pre-specified "lanes."	220	2	0
Total	355	33	28

obtained these numbers three years ago. Using our current system would have reduced the false positive rate, since most were due to simple infeasible paths that it can eliminate. (The severity of the errors made the given rate perfectly acceptable.)

3.2 Model Checking FLASH

While static analysis worked well on code-visible rules, it has difficulty with properties that were not visible in the source code, but rather implied by it, such as invariants over data structures or values produced by code operations. For example, that the sharing list for a dirty cache line is empty or that the count of sharing nodes equaled the number of caches a line was in. On the other hand, these sort of properties and FLASH structure in general are well-suited to model checking.

Unfortunately, the known hard problem with using model checking on real code is the need to write a specification (a "model") that describes what the software does. For example, it took a graduate student several months to build hand-written, heavily-simplified model of a single FLASH protocol [21]. Our model checking approach finessed this problem by using static analysis to automatically extract models from source code. We started the project after noticing the close correspondence between a hand-written specification of FLASH [21]) with the implementation code itself. FLASH code made heavy use of stylized macros and naming conventions. These "latent specifications" [20] made it relatively easy to pick out the code relevant to various important operations (message sends, interactions with the I/O subsystem, etc) and then automatically translate them to a checkable model.

Model checking with our system involves the following four steps. First, the user provides a *metal* extension that when run by our extensible compiler marks specific source constructs, such as all message buffer manipulations or sends. These extensions are essentially abstraction functions. Second, the system then

automatically extracts a backward slice of the marked code, as well as its dependencies. Third, the system translates the sliced code to a Murφ model. Fourth, the Murφ model checker checks the generated model along with a hand-written environment model.

Table 2 lists a representative subset of the rules we checked that static analysis would have had difficulty with. Surprisingly, there were relatively few errors in these properties as compared to the more shallow properties checked with static analysis.

Table 2. Description of a representative subset of invariants checked in four FLASH protocols using model checking. Checking these with static analysis would be difficult.

Invariants
The `RealPtrs` counter does not overflow (`RealPtrs` maintains the number of sharers).
Only a single master copy of each cache line exists (basic coherence).
A node can never put itself on the sharing list (sharing list is only for remote nodes).
No outstanding requests on cache lines that are already in `Exclusive` state.
Nodes do not send network messages to themselves.
Nodes never overflow their network queues.
Nodes never overflow their software queues (queue used to suspend handlers).
The protocol never tries to invalidate an exclusive line.
Protocol can only put data into the processor's cache in response to a request.

3.3 Myth: Model Checking Will Find More Bugs

The general perception within the bug-finding community is that since model checking is "deeper" than static analysis then if you take the time to model check code, you will find more errors. We have not found this to be true, either in this case study or in the next one. For FLASH, static analysis found roughly four times as many bugs as model checking, despite the fact that we spent more time on the model checking effort. Further, this differential was after we aggressively tried to increase bug counts. We were highly motivated to do so since we had already published a paper that found 34 bugs [2]; publishing a follow-on paper for a technique that required more work and found fewer was worrisome. In the end, only two of the eight bugs found with model checking had been missed by static analysis. Both were counter overflows that were deeper in the sense that it required a deep execution trace to find them. While they could potentially have been found with static analysis, doing so would have required a special-case checker.

The main underlying reason for the lower bug counts is simple: model checking requires running code, static analysis just requires you compile it. Model checking requires a working model of the environment. Environments are often messy and hard to specify. The formal model will simplify it. There were five

main simplifications that caused the model checker to miss FLASH bugs found with static analysis:

1. We did not model cache line data, though we did model the state that cache lines were in, and the actual messages that were sent. This omission both simplified the model and shrank the state space. The main implication in terms of finding errors was that there was nothing in the model to ensure that the data buffers used to send and receive cache lines were allocated, deleted or synchronized correctly. As a result, model checking missed 13 errors: all nine buffer allocation errors and all four buffer race conditions.
2. We did not model the FLASH I/O subsystem, primarily because it was so intricate. This caused the model checker to miss some of the message-length errors found by the static checker.
3. We did not model uncached reads or writes. The node controllers support reads and writes that explicitly bypass the cache, going directly to memory. These were used by rare paths in the operating system. Because these paths were rare it appears that testing left a relatively larger number of errors on them as compared to more common paths. These errors were found with static analysis but missed by the model checker because of this model simplification.
4. We did not model message "lanes." To prevent deadlock, the real FLASH machine divides the network into a number of virtual networks ("lanes"). Each different message type has an associated lane it should use. For simplicity, our model assumed no such restrictions. As a result, we missed the two deadlock errors found with static analysis.
5. FLASH code has many dual code paths — one used to support simulation, the other used when running on the actual FLASH hardware. Errors in the simulation code were not detected since we only checked code that would actually run on the hardware.

Taking a broader view, the main source of false negatives is not incomplete models, but the need to create a model at all. This cost must be paid for each new checked system and, given finite resources, it can preclude checking new code or limit checking to just code or properties whose environment can be specified with a minimum of fuss. A good example for FLASH is that time limitations caused us to skip checking the "`sci`" protocol, thereby missing five buffer management errors (three serious, two minor) found with static analysis.

3.4 Summary

Static analysis works well at checking properties that clearly map to source code constructs. Model checking can similarly leverage this feature to automatically extract models from source code.

As this case study shows, many abstract rules can be checked by small, simple static checkers. Further, the approach was effective enough to find errors in code that was (1) not written for verification and (2) had been heavily-tested for over five years.

After using both approaches, static analysis had two important advantages over model checking. First, in sharp contrast to the thorough, working code understanding demanded by model checking, static analysis allowed us to understand little of FLASH before we could check it, mainly how to compile it and a few sentences describing rules. Second, static analysis checks all paths in all code that you can compile. Model checking only checks code you can run; and of this code, only of paths you execute. This fact hurt its bug counts in the next case study as well.

4 Case Study: AODV

This section describes our experiences finding bugs in the AODV routing protocol implementation using both model checking and static analysis. We first describe CMC, the custom model checker we built, give an overview of AODV, and then compare the bugs found (and not found) by both approaches.

4.1 CMC Overview

While automatically slicing out a model for FLASH was far superior to hand constructing one, the approach had two problems. First, it required that the user have a intimate knowledge of the system, so that they could effectively select and automatically mark stand-alone subparts of it. Second, Murφ, like most modeling languages lacks many C constructs such as pointers, dynamic allocation, and bit operations. These omissions make general translation difficult.

We countered these problems by building CMC, a model checker that checks programs written in C [5]. CMC was motivated by the observation that there is no fundamental reason model checkers must use a weak input language. As it executes the implementation code directly, it removes the need to provide an abstract model, tremendously reducing the effort required to model check a system. As the implementation captures all the behaviors of the system, CMC is no longer restricted to behaviors that can be represented in conventional modeling languages. CMC is an explicit state model checker that works more or less like Murφ though it lacks many of Murφ's more advanced optimizations. As CMC checks a full implementation rather than an abstraction of it, it must handle much larger states and state spaces. It counters the state explosion problem by using aggressive approximate reduction techniques such as hashcompaction [22] and various heuristics [5] to slice unnecessary detail from the state.

4.2 AODV Overview

AODV (Ad-hoc On-demand Distance Vector) protocol [4] is a loop-free routing protocol for ad-hoc networks. It is designed to handle mobile nodes and a "best effort" network that can lose, duplicate and corrupt packets.

AODV guarantees that the network is always free of routing loops. If an error in the specification or implementation causes a routing loop to appear in

Table 3. Properties checked in AODV.

Assertion Type	Examples
Generic	Segmentation violations, memory leaks, dangling pointers.
Routing Loop	The routing tables of all nodes do not form a routing loop.
Routing Table	At most one routing table entry per destination. No route to self in the AODV-UU implementation. The hop count of the route to self is 0, if present. The hop count is either infinity or less than the number of nodes in the network.
Message Field	All reserved fields are set to 0. The hop count in the packet can not be infinity.

Table 4. Lines of implementation code vs. CMC modeling code.

Protocol	Checked Code	Correctness Specification	Environment network	Environment stubs	State Canonicalization
mad-hoc	3336	301	400	100	165
Kernel AODV	4508	301	400	266	179
AODV-UU	5286	332	400	128	185

the network, the protocol has no mechanism to detect or recover from them, allowing the loop to persist forever, completely breaking the protocol. Thus, it is crucial to comprehensively test both the AODV protocol specification and any AODV implementation for loop freeness as thoroughly as possible.

AODV is relatively easy to model check. Its environmental model is greatly simplified by the fact that the only input it deals with are user requests for a route to a destination. This can be easily modeled as a nondeterministic input that is enabled in all states. Apart from this, an AODV node responds to two events, a timer interrupt and a packet received from other AODV nodes in the network. Both are straightforward to model.

4.3 Model Checking AODV with CMC

We used CMC to check three publicly-available AODV implementations: *mad-hoc* (Version 1.0) [23], *Kernel AODV* (Version 1.5) [24], and *AODV-UU* (Version 0.5) [25]. While it is not clear how well these implementations are tested, they have been used in different testbeds and network simulation environments [26]. On average, the implementations contain 6000 lines of code.

For each implementation, the model consists of a core set of unmodified files. This model executes along with an environment which consists of a network model and simplified implementations (or "stubs") for the implementation functions not included in the model. Table 4 describes the model and environment for these implementations. All three models reuse the same network model.

Table 5. Number of bugs of each type in the three implementations of AODV. The figures in parenthesis show the number of bugs that are instances of the same bug in the mad-hoc implementation.

	mad-hoc	**Kernel AODV**	**AODV-UU**
Mishandling `malloc` failures	4	6	2
Memory Leaks	5	3	0
Use after free	1	1	0
Invalid Routing Table Entry	0	0	1
Unexpected Message	2	0	0
Generating Invalid Packets	3	2 (2)	2
Program Assertion Failures	1	1 (1)	1
Routing Loops	2	3 (2)	2 (1)
Total	18	16 (5)	8 (1)
LOC per bug	185	285	661

Table 6. Comparing static analysis (MC) and CMC. Note that for the MC results we only ran a set of generic memory and pointer checkers rather than writing AODV-specific checkers. Generating the MC results took about two hours, rather than the weeks required for AODV.

		Bugs Found		
		CMC & MC	**CMC alone**	**MC alone**
Generic Properties	Mishandling `malloc` failures	11	1	8
	Memory Leaks	8	-	5
	Use after free	2	-	-
Protocol Specific	Invalid Routing Table Entry	-	1	-
	Unexpected Message	-	2	-
	Generating Invalid Packets	-	7	-
	Program Assertion Failures	-	3	-
	Routing Loops	-	7	-
	Total	21	21	13

As CMC was being developed during this case study, it is difficult to gauge the time spent in building these models as opposed to building the model checker itself. As a rough estimation, it took us two weeks to build the first, mad-hoc model. Building subsequent models was easier, and it took us one more week to build both these models.

Table 3 describes the assertions CMC checked in the AODV implementations. CMC automatically checks certain generic assertions such as segmentation violations. Additionally, the protocol model checks that routing tables are loop free at all instants and that each generated message and route inserted into the table obey various assertions. Table 4 gives the lines of code required to add these correctness properties.

CMC found a total of 42 errors. Of these, 35 are unique errors in the implementations and one is an error in the underlying AODV specification. Table 5

summarizes the set of bugs found. The Kernel AODV implementation has 5 bugs (shown in parentheses in the table) that are instances of the same bug in mad-hoc. The AODV specification bug causes a routing loop in all three implementations.

4.4 Static AODV Checking: More Paths + More Code = More Bugs

We also did a cursory check of the AODV implementations using a set of static analysis checkers that looked for generic errors such as memory leaks and invalid pointer accesses. The entire process of checking the three implementations and analyzing the output for errors took two hours. Static analysis found a total of 34 bugs.

Table 6 compares the bugs found by static analysis and CMC. It classifies the bugs found into two broad classes depending on the properties violated: generic and protocol specific. For generic errors, our results matched those in the FLASH case study: static analysis found many more bugs than model checking. Except for one, static analysis found all the bugs that CMC could find. As in our previous case study (§3.3), the fundamental reason for this difference is that static analysis can check all paths in all code that you can compile. In contrast, model checking can only check code triggered by the specific environment model. Of the 13 errors not found by CMC, 6 are in parts of the code that are either not included in the model or cut out during environment modeling. For instance, static analysis found two cases of mishandled `malloc` failures in multicast routing code. All our CMC models omitted this code.

Additionally, CMC missed errors because of subtle mistakes in its environment model. For example, the mad-hoc implementation uses the `send_datagram` function to send a packet and has a memory leak when the function fails. However, our environment erroneously modeled the `send_datagram` as always succeeding. CMC thus missed this memory leak. Such environmental errors caused CMC to miss 6 errors in total. Static analysis found one more more error in dead code that can never be executed by any CMC model.[2]

4.5 Where Model Checking Won: More Checks = More Bugs

In the class of protocol-specific errors, CMC found 21 errors while static analysis found none. While this was partly because we did not check protocol-specific properties, many of the errors would be difficult to find statically. We categorize the errors that were found my model checking but missed by static analysis into three classes and describe them below.

By executing code, a model checker can check for properties that are not easily visible to static inspection (and thus static analysis). Many protocol specific properties fall in this class. Properties such as deadlocks and routing loops

[2] Static analysis also found a null pointer violation in one of our environment models! We do not count this error.

involve invariants of objects across multiple processes. Detecting such loops statically would require reasoning about the entire execution of the protocol, a difficult task. Also, present static analyzers have difficulty analyzing properties of heap objects. The error in Figure 1 is a good example. This error requires reasoning about the length of a linked list, similar to many heap invariants that static analyzers have difficulty with. Here, the code attempts to allocate `rerrhdr_msg.dst_cnt` temporary message buffers. It correctly checks for `malloc` failure and breaks out of the loop. However, it then calls `rec_rerr` which contains a loop that assumes that `rerrhdr_msg.dst_cnt` list entries were indeed allocated. Since the list has fewer entries than expected, the code will attempt to use a null pointer and get a segmentation fault.

```
// aodv_deamon.c:aodv_recv_message
   for(rerri=0; rerri<rerrhdr_msg.dst_cnt;rerri++) {
      // Step 1: break with < dst_cnt elements in rerrhdr_msg list.
      if (!(tp = malloc(...)))
         break;
      tp->next = rerrhdr_msg.unr_dst;   // enqueue onto list.
      rerrhdr_msg.unr_dst = tp;
      ...
   }
   // Step 2: rec_rerr assumes dst_cnt elements in rerrhdr_msg
  rec_rerr(&info_msg, &rerrhdr_msg);
  ...
int rec_rerr(struct info *tmp_info, struct rerrhdr *rh) {
   ...
   // Step 3: iterates rh->dst_cnt times even if not that many elements
   for(i = 0, t = rh->unr_dst; i < rh->dst_cnt; i++, tp = tp->next) {
     // ERROR: tp can be null!
     tmp_rtentry = getentry(tp->unr_dst_ip);
```

Fig. 1. The one memory error missed by static analysis: requires reasoning about values, and is a good example of where model checking can beat static analysis.

A second advantage model checking has is that it checks for actual errors, rather than having to reason about all the different ways the error could be caused. If it catches a particular error type it will do so no matter the cause of the error. For example, a model checker such as CMC that runs code directly will detect all null pointer dereferences, deadlocks, or any operation that causes a runtime exception since the code will crash or lock up. Importantly, it will detect them without having to understand and anticipate all the ways that these errors could arise. In contrast, static analysis cannot do such end-to-end-checks, but must instead look for specific ways of causing a given error. Errors caused by actions that the checker does not know about or cannot analyze will not be flagged. and so minimize false positives by looking for errors only in specific analyzable contexts.

A good example is the error CMC found in the AODV specification, shown in Figure 2. This error arises because the specification elides a check on the sequence number of a received packet. Here, the node receives a packet with a stale route with an old sequence number. The code (and the specification) erroneously updates the sequence number of the current route without checking if the route in the packet is valid. This results in a routing loop. Once the cause of the routing loop is known, it is possible (and easy) to statically ensure that all sequence number updates to the routing table from any received packets involve a validity check. However, there are only a few places where such specialized checks can be applied, making it hard to recoup the cost of writing the checker. Moreover, exhaustively enumerating all different causes for a routing loop is not possible. On the other hand, a model checker can check for actual errors without the need for reasoning about their causes.

In a more general sense, model checking's end-to-end checks mean it can give guarantees much closer to total correctness than static analysis can. No one would be at all surprised if code that passed all realistic static checks immediately crashed when actually run. On the other hand, given a good environment model and input sequences, it is much more likely that model checked code actually works when used. Because model checking can verify the code was actually totally *correct* on the executions that it tested, then if state reduction techniques allow these executions to cover much of the initial portion of the search space, it will be difficult for an implementation to efficiently get into new, untested areas. At risk of being too optimistic this suggests that even with a residual of bugs in the model checked implementation it will be so hard to trigger them that they are effectively not there.

```
// madhoc:rerr.c:rec_rerr
//    recv_rt: route we just received from network.
//    cur_rt: current route entry for same IP address.
cur_rt = getentry(recv_rt->dst_ip);
if(cur_rt != NULL && ...) {
      // Bug: updates sequence number without checking that
      // received packet newer than route table entry!
      cur_rt->dst_seq = recv_rt->dst_seq;
```

Fig. 2. The AODV specification error. A common pattern: this bug could have been caught with static analysis, but there were so few places to check that it would be difficult to recover the overhead of building checker.

A final model checking advantage was that there were true bugs that both methods would catch, but because they "could not happen" would be labeled as false positives when found with static analysis. In contrast, because model checking produces an execution trace to the bug, they would be correctly labeled. The best example was a case where an AODV node receives a "route response" packet in reply to a "route request" message it has sent. The code first looked

up the route for the reply packet's IP address and then used this route table entry without checking for null. While a route table lookup can return null in general, this particular lookup "cannot be null," since a node only sends route requests for valid route table entries. If this unchecked dereference was flagged with static analysis it would be labeled a false positive. However, if the node (1) sends this message, (2) reboots, and (3) receives the response the entry can be null. Because model checking gave the exact sequence of unlikely events the error was relatively clear.

4.6 Summary

The high bit for AODV: model checking hit more properties, but static hit more code; when they checked the same property static won. The latter was surprising since it implies that most bugs were shallow, requiring little analysis, perhaps because code difficult for the analysis to understand is similarly hard for programmers to understand. As with FLASH, the difference in time was significant: hours for static versus weeks for model checking.

One view of the trade-off between the approaches is that static analysis checks code well, but checks the implications of code relatively poorly. On the other hand, model checking checks implications relatively better, but because of its its problems with abstraction and coverage, can be less effective checking the actual code itself.

These results suggest that while model checking can get good results on real systems code, in order to justify their significant additional effort they must target properties not checkable statically.

5 Case Study: TCP

This section describes our efforts in model checking the Linux TCP implementation. We decided to check TCP after the relative success of AODV since it was the hardest code we could think of in terms of finding bugs. There were several sources of difficulty. First, the version we checked (from Linux 2.4.19) is roughly ten times larger than AODV code (50K lines versus 6K). Second, it is mature, often frequently audited code. Third, since almost all Linux sites constantly use TCP, it is one of the the heaviest-tested pieces of open source code around.

We had expected TCP would require only modest more effort than AODV. As Section 5.1 describes below, this expectation was wildly naive. Further, as Section 5.2 shows, it is a very different matter to get code to run at all and getting it to run so that you can comprehensively test it.

5.1 The Environment Problem: Lots of Time, Lots of False Positives

The system to be model checked is typically present in a larger execution context. For instance, the Linux TCP implementation is present in the Linux kernel, and closely interacts with other kernel modules. Before model checking, it is

necessary to extract the relevant portions of the system to be model checked and create an appropriate *environment* that allows the extracted system to run stand alone. This environment should contain *stubs* for all external functions that the system depends on. With an implementation-level model checker like CMC, this process is very similar to building a harness for unit testing. However, building a comprehensive environment model for a large and complex system can be difficult. This difficulty is known to an extent in the model checking literature, but is typically underplayed.

Extracting code amounts to deciding for each external function that the system calls, whether the function should be included in the checked code base, or to instead create a stub for the function and include the stub in the environment model. The advantages of including the function in the checked code are (1) the model checker executes the function and thus can potentially find errors in it (2) there is no need to create the stub or to maintain the stub as the code evolves. However, including an external function in the system has two downsides: (1) this function can potentially increase the state space and (2) it can call additional functions for which stubs need to be provided.

Conventional wisdom dictates that one cut along the narrowest possible interface. The idea is that this requires emulating the fewest possible number of functions, while minimizing the state space. However, while on the surface this makes sense, it was an utter failure for TCP. And, as we discuss below, we expect it to cause similar problems for any complex system.

Failure: building a kernel library. Our first attempt to model check TCP did the obvious thing: we cut out the TCP code proper along with a few tightly coupled modules such as IP and then tried to make a "kernel library" that emulated all the functions this code called. Unfortunately, TCP's comprehensive interaction with the rest of the kernel meant that despite repeated attempts and much agonizing we could only reduce this interface down to 150 functions, each of which we had to write a stub for.

We abandoned this effort after months of trying to get the stubs to work correctly. We mainly failed because TCP, like other large complex subsystems, has a large, complex, messy interface its host system. Writing a harness that perfectly replicates the *exact* semantics of this (often poorly documented) interface is difficult. In practice these stubs have a myriad of subtle corner-case mistakes. Since model checkers are tuned to find inconsistencies in corner cases, they will generate a steady stream of bugs, all false. To make matters worse, these false positives tend to be much harder to diagnose than those caused by static analysis. The latter typically require seconds or, rarely, several minutes to diagnose. In contrast, TCP's complexity meant that we could spend *days* trying to determine if an error report was true or false. One such example: an apparent TCP storage leak of a socket structure was actually caused by an incorrect stub implementation of the Linux timer model. The TCP implementation uses a function `mod_timer()` to modify the expiration time of a previously queued timer. This function's return value depends on whether the timer is pending when the

function is called. Our initial stub always returned the same value. This mistake confused the reference counting mechanism of the socket structures in an obscure way, causing a memory leak, which took a lot of manual examination to unravel.

It is conceivable that with more work we could have eventually replicated all 150 functions in the interface to perfection (at least until their semantics changed). However, we did not seem to be reaching a fixed point — in fact, each additional false positive seemed harder to diagnose. In the end, it was easier to do the next approach.

Surprising success: run Linux in CMC. While it seems intuitive that one should cut across the smallest interface point, it is also intuitive that one cut along well-defined and documented interfaces. It turns out that for TCP (and we expect for complex code in general) the latter is a better bet. It greatly simplifies the environment modeling. Additionally, it makes it less likely that these interfaces will change in future revisions, allowing the same environment model to be (re)used as the system implementation evolves. While the approach may force the model checker to deal with larger states and state spaces, the benefits of a clean environment model seem to outweigh the potential disadvantage.

It turns out that for TCP there are only two very well-defined interfaces: (1) the system call interface that defines the interaction between user processes and the kernel and (2) the "hardware abstraction layer" that defines the interaction between the kernel and the architecture. Cutting the code at this level means that we wind up pulling the entire kernel into the model! While initially this sounded daunting, in practice it turned out to not be that difficult to "port" the kernel to CMC by providing a suitable hardware abstraction layer. This ease was in part because we could reuse a lot of work from the User Mode Linux (UML) [27]) project, which had to solve many of the same problems in its aim to run a working kernel as a user process.

In order to check the TCP implementation for protocol compliance, we wrote a TCP reference model based on the TCP RFC. CMC runs this alongside the implementation, providing it with the same inputs as to the implementation, and reports if their states are inconsistent as a protocol violation error.

5.2 The Coverage Problem: No Execute, No Bug

As with dynamic checking tools, model checking can only find errors on executed code paths. In practice it is actually quite difficult to exercise large amounts of code. This section measures how comprehensively we could check TCP.

We used two metrics to measure coverage. The first is line coverage of the implementation achieved during model checking. While the crudeness of this measure means it may not correspond to how well the system has been checked, it does effectively detect the parts that have *n*ot been tested. The second is "protocol coverage," which corresponds to the abstract protocol behaviors tested by the model checker. We calculate protocol coverage as the line coverage achieved in the TCP reference model mentioned above. This roughly represents the degree to which the abstract protocol transitions have been explored.

Table 7. Coverage achieved during model refinement. The branching factor is a measure of the state space size.

Description	Line Coverage	Protocol Coverage	Branching Factor	Bugs
Standard server and client	47.4 %	64.7 %	2.91	2
+ simultaneous connect	51.0 %	66.7 %	3.67	0
+ partial close	52.7 %	79.5 %	3.89	2
+ message corruption	50.6 %	84.3 %	7.01	0
Combined Coverage	55.4 %	92.1 %		

We used the two metrics to detect where we should make model checking more comprehensive. Low coverage often helped in pointing out errors in our environment model. Table 7 gives the coverage achieved with each step in the model refinement process. We measured coverage cumulatively using three search techniques: breadth-first, depth-first, and random. In random search, each generated state is given a random priority. Table 7 also reports the branching factor of the state space as a measure of its size — lower branching factors are good, since they mean the state increases exponentially less each step in. For the first three models the branching factor is calculated from the number of states in the queue at depth 10 during a breadth first search. For the fourth model, CMC ran out of resources at depth 8, and the branching factor is calculated at this depth.

The first model consists of a single TCP client communicating with a single TCP server. Once the connection is established, the client and server exchange data in both directions before closing the connection. This standard model discovered two protocol compliance bugs in the TCP implementation. The second model adds multiple simultaneous connections, which are initiated nondeterministically. The third model lets either end of the connection nondeterministically decide to close it during data transfer. This improved coverage and resulted in the discovery of two more errors. Finally, much of the remaining untested functionality was code to handle bad packets, so we corrupted packets by nondeterministically toggling selected key control flags in the TCP packet. While these corrupted packets triggered a lot of recovery code they also resulted in an enormous increase in the state space. Tweaking the environment the right way to achieve a more effective search still remains an interesting but unsolved problem.

In the end we detected four errors in the Linux TCP implementation. All are instances where the implementation fails to meet the TCP specification. These errors are fairly complex and require an intricate sequence of events to trigger the error.

6 Conclusion

This paper has described trade-offs between both static analysis and model checking, as well as some of the surprises we encountered while applying model checking to large software systems. Neither static analysis nor model checking are at the stage where one dominates the other. Model checking gets more properties, but static analysis hit more code; when they checked the same property static analysis won.

The main advantages static analysis has over model checking: (1) gets all paths in all code that can compile, rather than just executed paths in code you can run, (2) only requires a shallow understanding of code, (3) applies in hours rather than weeks, (4) easily checks millions of lines of code, rather than tens of thousands, (5) can find thousands of errors rather than tens. The first question you ask with static analysis is "how big is the code?" Nicely, bigger is actually better, since it lets you amortize the fixed cost of setting up checking. Model checking's first question is "what does the code do?" This is both because many program classes cannot be model checked and because doing so requires an intimate understanding of the code. Finally, given enough code we are surprised when static analysis gets no results, but less surprised if model checking does not (or if the attempt abandoned). Most of these are direct implications of the fact that model checking runs code and static analysis does not.

We believe static analysis will generally win in terms of finding as many bugs as possible. In this sense it is better, since less bugs gets users closer to the desired goal of the absence of bugs ("total correctness"). However, model checking has advantages that seem hard for static analysis to match: (1) it can check the implications of code, rather than just surface-visible properties, (2) it can do end-to-end checks (the routing table has no loops) rather than having to anticipate and craft checks for all ways that an error type can arise, (3) it gives much stronger correctness results — we would be surprised if code crashed after being model checked, whereas we are not surprised at all if it crashes after being statically checked.

A significant model checking drawback is the need to create a working, correct environment model. We had not realized just how difficult this would be for large code bases. In all cases it added weeks to months of effort compared to static analysis. Also, practicality forced omissions in model behavior, both deliberate and accidental. In both FLASH and AODV, unmodeled code (such as omitting the I/O system or multicast support) led to many false negatives. Finally, because the model must perfectly replicate real behavior, we had to fight with many (often quite-tricky-to-diagnose) false positives during development. In TCP this problem eventually forced us to resort to running the entire Linux kernel inside our model checker rather than creating a set of fake stubs to emulate TCP's interface to it. This was not anticipated.

All three model checking case studies reinforced the following four lessons:

1. No model is as good as the implementation itself. Any modification, translation, approximation done is a potential for producing false positives, danger of checking far less system behaviors, and of course missing critical errors.

2. Any manual work required in the model checking process becomes immensely difficult as the scale of the system increases. In order to scale, model checker should require as little user input, annotations and guidance as possible.
3. If an unit-test framework is not available, then define the system boundary only along well-known, public interfaces.
4. Try to cover as much as possible: the more code you trigger, the more bugs you find, and more useful model checking is.

Acknowledgments.
This paper recounts research done with others. In particular, we thank David Lie and Andy Chou for their discussions of lessons learned model checking the FLASH code (of which they did the bulk of the work) and David Park for his significant help developing CMC and model-checking TCP. We especially thank David Dill for his valuable discussions over the years. We thank Willem Visser for thoughtful comments on a previous version of this paper.

This research was supported in part by DARPA contract MDA904-98-C-A933, by GSRC/MARCO Grant No:SA3276JB, and by a grant from the Stanford Networking Research Center. Dawson Engler is partially supported by an NSF Career Award.

References

1. Kuskin, J., Ofelt, D., Heinrich, M., Heinlein, J., Simoni, R., Gharachorloo, K., Chapin, J., Nakahira, D., Baxter, J., Horowitz, M., Gupta, A., Rosenblum, M., Hennessy, J.: The Stanford FLASH multiprocessor. In: Proceedings of the 21st International Symposium on Computer Architecture. (1994)
2. Chou, A., Chelf, B., Engler, D., Heinrich, M.: Using meta-level compilation to check FLASH protocol code. In: Ninth International Conference on Architecture Support for Programming Languages and Operating Systems. (2000)
3. Lie, D., Chou, A., Engler, D., Dill, D.: A simple method for extracting models from protocol code. In: Proceedings of the 28th Annual International Symposium on Computer Architecture. (2001)
4. C.Perkins, Royer, E., Das, S.: Ad Hoc On Demand Distance Vector (AODV) Routing. IETF Draft, http://www.ietf.org/internet-drafts/draft-ietf-manet-aodv-10.txt (2002)
5. Musuvathi, M., Park, D., Chou, A., Engler, D.R., Dill, D.L.: CMC: A Pragmatic Approach to Model Checking Real Code. In: Proceedings of the Fifth Symposium on Operating Systems Design and Implementation. (2002)
6. Dill, D.L., Drexler, A.J., Hu, A.J., Yang, C.H.: Protocol verification as a hardware design aid. In: IEEE International Conference on Computer Design: VLSI in Computers and Processors. (1992) 522–525
7. Holzmann, G.J.: The model checker SPIN. Software Engineering **23** (1997) 279–295
8. Corbett, J., Dwyer, M., Hatcliff, J., Laubach, S., Pasareanu, C., Robby, Zheng, H.: Bandera: Extracting finite-state models from Java source code. In: ICSE 2000. (2000)

9. Holzmann, G., Smith, M.: Software model checking: Extracting verification models from source code. In: Invited Paper. Proc. PSTV/FORTE99 Publ. Kluwer. (1999)
10. Godefroid, P.: Model Checking for Programming Languages using VeriSoft. In: Proceedings of the 24th ACM Symposium on Principles of Programming Languages. (1997)
11. Chandra, S., Godefroid, P., Palm, C.: Software model checking in practice: An industrial case study. In: Proceedings of International Conference on Software Engineering (ICSE). (2002)
12. Brat, G., Havelund, K., Park, S., Visser, W.: Model checking programs. In: IEEE International Conference on Automated Software Engineering (ASE). (2000)
13. Bush, W., Pincus, J., Sielaff, D.: A static analyzer for finding dynamic programming errors. Software: Practice and Experience **30** (2000) 775–802
14. Das, M., Lerner, S., Seigle, M.: Esp: Path-sensitive program verification in polynomial time. In: Conference on Programming Language Design and Implementation. (2002)
15. Flanagan, C., Leino, M.R.K., Lillibridge, M., Nelson, C., Saxe, J., Stata, R.: Extended static checking for Java. In: 2002 Conference on Programming Language Design and Implementation. (2002) 234–245
16. Chen, H., Wagner, D.: MOPS: an infrastructure for examining security properties of software. In: Proceedings of the 9th ACM conference on Computer and communications security, ACM Press (2002) 235 – 244
17. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: Proceedings of the SIGPLAN '01 Conference on Programming Language Design and Implementation. (2001)
18. Engler, D., Chelf, B., Chou, A., Hallem, S.: Checking system rules using system-specific, programmer-written compiler extensions. In: Proceedings of Operating Systems Design and Implementation (OSDI). (2000)
19. Hallem, S., Chelf, B., Xie, Y., Engler, D.: A system and language for building system-specific, static analyses. In: SIGPLAN Conference on Programming Language Design and Implementation. (2002)
20. Engler, D., Chen, D., Hallem, S., Chou, A., Chelf, B.: Bugs as deviant behavior: A general approach to inferring errors in systems code. In: Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles. (2001)
21. Park, S., Dill, D.: Verification of FLASH cache coherence protocol by aggregation of distributed transactions. In: Proceedings of the 8th ACM Symposium on Parallel Algorithsm and Architectures. (1996) 288–296
22. Stern, U., Dill, D.L.: A New Scheme for Memory-Efficient Probabilistic Verification. In: IFIP TC6/WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification. (1996)
23. Lilieblad, F., et.al.: Mad-hoc AODV Implementation (http://mad-hoc.flyinglinux.net/)
24. Klein-Berndt, L., et.al.: Kernel AODV Implementation. (http://w3.antd.nist.gov/wctg/aodv_kernel/)
25. *et al.*, E.N.: AODV-UU Implementation. (http://user.it.uu.se/ henrikl/aodv/)
26. Lundgren, H., Lundberg, D., Nielsen, J., Nordstrom, E., Tschudin, C.: A large-scale testbed for reproducible ad hoc protocol evaluations. In: IEEE Wireless Communications and Networking Conference. (2002)
27. (The User-mode Linux Kernel) http://user-mode-linux.sourceforge.net/.

Automatic Inference of Class Invariants

Francesco Logozzo

STIX - École Polytechnique
F-91128 Palaiseau, France
Francesco.Logozzo@polytechnique.fr

Abstract. We present a generic framework for the automatic and modular inference of sound class invariants for class-based object oriented languages. The idea is to derive a sound class invariant as a conservative abstraction of the class semantics. In particular we show how a class invariant can be characterized as the solution of a set of equations extracted from the program source. Once a static analysis for the method bodies is supplied, a solution for the former equation system can be iteratively computed. Thus, the class invariant can be automatically inferred. Moreover, our framework is modular since it allows the derivation of class invariants without any hypothesis on the instantiation context and, in the case of subclassing, without accessing to the parent code.

1 Introduction

A class is correct or incorrect not by itself, but with respect to a specification. For instance, a specification can be the absence of runtime errors, such as null-pointers dereference or the absence of uncaught exceptions. More generally, a specification can be expressed in a suitable formal language. The software engineering community [14] proposes to annotate the source code with class invariants, method preconditions and postconditions in order to specify the *desired* behavior of the class. A class invariant is a property valid for each instance of the class, before and after the execution of any method, in any context. A method precondition is a property satisfied before the execution of the method and a postcondition is a property valid just after its execution.

The natural question with such an approach is: *"Does the class respect its specification?"*. The traditional approach is to monitor the assertions, so that for instance the preconditions and the class invariant are checked before the execution of a method. Such an approach has many drawbacks. For example, it requires checking arbitrary complex assertions so that it may introduce a non-negligible slowdown at runtime. Moreover it is inherently not sound. In fact the code must be executed in order to test if an assertion is violated or not. However, program execution or testing can only cover finitely many test cases so that the global validity of the assertion cannot be proved. Therefore the need for formal methods arises.

The approach based on abstract interpretation consists in computing an approximation of the class semantics and then to check whether it satisfies the

B. Steffen and G. Levi (Eds.): VMCAI 2004, LNCS 2937, pp. 211–222, 2004.

specification. In particular if a sound class invariant, which is inferred from the program source, matches the specification then the class itself matches the specification (because of soundness). Therefore a static analyzer capable of inferring sound class invariants can be used as an effective verification tool [3]. Furthermore class invariants can be used to optimize the compiled-code, e.g. to drop superfluous exception handlers or synchronizations, and for code documentation.

An effective static analysis framework for object oriented languages must take into account three issues:

- the different instantiation contexts of a class (*context*-modularity);
- the presence in the class to be verified of references to classes whose code is not available (*has-a*-modularity);
- the specialization through inheritance of a class whose source code is not available (*is-a*-modularity).

Our Work. We present a generic framework for the automatic and modular inference of sound class invariants for class-based object oriented languages.

Our framework is highly generic: it is language independent and more important any abstract domain can be plugged in. As static analyses consider particular properties (e.g. pointer aliasing or linear relationships betweens variables) the choice of a particular analysis influences the property reflected by the so-inferred class invariant. And hence it influences the check of the program specification. For instance, if the specification is, in some formal language, *"The class never raises the* `null`*-pointer exception"* then we are likely to instantiate the framework with a pointer analysis, in order to show that the methods in the class never cause the throwing of such an exception.

The framework is fully *context*-modular, as a class can be analyzed aside from the context in which it is used. It is *has-a*-modular, because if a class `C` references a class `H` (e.g. `C` contains a variable of type `H` or a method of `C` calls a function of `H`) then the `H` invariant can replace the `H` source code. Furthermore it is *is-a*-modular because a subclass can be analyzed by just accessing the superclass invariant, and *not* its source code.

Related Work. Several static analyses have been developed for object oriented languages as e.g. [2] and some of them have modular features e.g. [4] in that they analyze a program fragment without requiring the full program to stay in memory. Nevertheless they are different from the present work in that none of them is able to discover class invariants, essentially because they do not take into account peculiarities of object oriented languages such as the state encapsulation.

Some other works can infer class invariants w.r.t. to a particular property [1, 17,8,9]. Our work is more general than these. For instance, we consider arbitrary properties, we do not require any human interaction (unlike [17,9]) and we deal with the problem of computing subclass invariants without seeing the parent's code. Moreover, our approach is based on a conservative approximation of the class semantics so, unlike [8], it is sound.

In a previous work [12] we introduced a different approach for the analysis of object oriented languages, based on the use of *symbolic relations* and class-to-

```
1  class StackError extends Exception {}
2
3  public class Stack {
4    // Inv : 1 <= size, 0 <= pos <= size, size=stack.length
5
6    protected int size, pos;
7    protected Object[] stack;
8
9    Stack(int size) {
10     this.size = Math.max(size,1); this.pos  = 0;
11     this.stack = new Object[this.size];
12   }
13
14   boolean isEmpty() { return (pos <= 0);    }
15   boolean isFull()  { return (pos >= size); }
16
17   Object top() throws StackError {
18      if (!isEmpty())
19           // 0 <= pos-1 < stack.length
20         return stack[pos-1];
21      else throw new StackError();
22   }
23   void push(Object o) throws StackError {
24     if(!isFull()) {
25          // 0 <= pos < size
26        stack[pos++] = o;
27     } else throw new StackError();
28   }
29   void pop() throws StackError {
30     if(!isEmpty())
31        pos--;
32     else throw new StackError();
33   }
34 }
```

Fig. 1. Java source for the Stack class.

class transformations. The main novelty of the present work is the handling of inheritance. In fact, we introduce a generic framework for the modular inference of class invariants, in which arbitrary abstract domains can be plugged-in. Then, we show how it can be used for inferring subclass invariants without accessing to the parent code. To the best of our knowledge the present paper is the first one to address the problem of *is-a*-modular analyses.

2 Examples

We will illustrate the results of this paper through the examples in Fig. 1 and Fig. 2. The first class, `Stack`, is the implementation of a stack parameterized by its size, specified at object creation time. It provides methods for pushing and popping elements as well as testing if the stack is empty or full. Moreover, as the internal representation of the stack is hidden a stack object can only be manipulated through its methods. The second class, `StackWithUndo` extends the first one by adding to the stack the capability of performing the *undo* of the last operation.

The comments in the figures are automatically derived when the framework presented in this paper is instantiated with the Octagon abstract domain [15]. In particular, for the `Stack` we have been able to discover the class invariant `Inv` without any hypothesis on the class instantiation context so the analysis is *context*-modular. The invariant `Inv` guarantees that the array `stack` is never accessed out of its boundaries. This implies that the out-of-bounds exception is never thrown (verification) so that the bounds checks at lines 20 and 26 can be omitted from the generated bytecode (optimization).

The class invariant for `StackWithUndo`, `SubInv`, states that the parent class invariant is still valid and moreover the field `undoType` cannot assume values outside the interval $[-1, 1]$. It implies that the method `undo` will never raise the exception `StackErr`. Once again this information can be used for verification (if a class never raises an exception `Exc`, then the exceptional behavior described by `Exc` is never shown) and for optimization (as the exception handling can be dropped). Finally it is worth noting that `SubInv` has been obtained without accessing to the parent code but just to its class invariant, thus in an *is-a*-modular fashion.

3 Syntax and Concrete Semantics

The concrete semantics describes the properties of the execution of programs. The goal of a static analysis is to provide an effective computable approximation of the concrete semantics [5]. Therefore, the first step for the design of a static analysis is the definition of the concrete semantics.

A class is a template for objects. It is provided by the programmer that specifies the fields, the methods and the class constructor. Then it can be abstractly modeled by a triple, as in the following definition:

Definition 1. *A class* $\mathtt{C}$ *is a triple* $\langle \mathtt{F}, \mathtt{init}, \mathtt{M} \rangle$ *where* $\mathtt{F}$ *is a set of distinct variables,* $\mathtt{init}$ *is the class constructor and* $\mathtt{M}$ *is a set of function definitions.*

It is worth noting that for the sake of generality in the previous definition we did not ask to have typed fields or methods. Moreover we assume that all the class fields are protected. This is just to simplify the exposition, and it does not cause any loss of generality: in fact any external access to a field `f` can be simulated by a couple of methods `set_f`/ `get_f`.

Given a class $\mathtt{C} = \langle \mathtt{F}, \mathtt{init}, \mathtt{M} \rangle$, every instance of $\mathtt{C}$ has an internal state $\sigma \in \Sigma$ that is a function from fields to values, i.e. $\Sigma = [\mathtt{F} \to \mathsf{D}_{\mathsf{val}}]$.

When a class is instantiated, for example by means of the `new` construct, the class constructor is called to set up the new object internal state. This can be modeled by a semantic function $\mathsf{i}[\![\mathtt{init}]\!] \in [\mathsf{D}_{\mathsf{in}} \to \wp(\Sigma)]$, where D_{in} is the semantic domain for the constructor input values (if any). We consider sets in order to model non-determinism, e.g. user input.

The semantics of a method $\mathtt{m}$ is a function $\mathsf{m}[\![\mathtt{m}]\!] \in [\mathsf{D}_{\mathsf{in}} \times \Sigma \to \wp(\mathsf{D}_{\mathsf{out}} \times \Sigma)]$. Indeed a method is called with two parameters: the method actual parameters and the internal state of the object it belongs to. The output of a method is

```
class StackWithUndo extends Stack {

  // SubInv : Inv,  -1 <= undoType <= 1,
  //          if undoType ==  1 then  0 < pos
  //     else if undoType == 0  then  0 <= pos <= size
  //     else if undoType == -1 then  pos < size

  protected Object undoObject;
  protected int undoType;

  StackWithUndo(int x) {
    super(x);
    undoType = 0; undoObject = null;
  }
  void push(Object o) throws StackError {
    undoType   = 1;
    super.push(o);
  }
  void pop() throws StackError {
    if(!isEmpty()) {
       undoType   = -1;
       undoObject = stack[pos-1];
     }
    super.pop();
  }
      // StackError never thrown
  void undo() throws StackError {
    if(undoType == -1) {
      super.push(undoObject);
      undoType = 0;
    } else if (undoType == 1) {
      super.pop();
      undoType = 0;
    }
  }
}
```

Fig. 2. Stack class extension with undo capabilities.

a set of pairs $\langle$ return value (if any), new object state $\rangle$. The set of the initial states is:

$$S_0 = \{\sigma \mid \exists v \in \mathsf{D_{in}}.\ \sigma \in \mathsf{i}[\![\texttt{init}]\!](v)\},$$

and the method collecting semantics $\mathsf{M}[\![\mathtt{m}]\!] \in [\wp(\Sigma) \to \wp(\Sigma)]$:

$$\mathsf{M}[\![\mathtt{m}]\!](S) = \{\sigma' \in \Sigma \mid \exists \sigma \in S.\ \exists v \in \mathsf{D_{in}}.\ \exists v' \in \mathsf{D_{out}}.\ \langle v', \sigma' \rangle \in \mathsf{m}[\![\mathtt{m}]\!]\langle v, \sigma \rangle\}.$$

Then the class reachable states, $\mathsf{c}[\![\mathtt{C}]\!]$, are given by the least solution, w.r.t. set inclusion, of the following recursive equation, where n is the number of methods of C:

$$S = S_0 \cup \bigcup_{i=1}^{n} \mathsf{M}[\![\mathtt{m}_i]\!](S). \tag{1}$$

The former equation characterizes, according to the intuition, the set of states that are reachable before and after the execution of any method in any instance

of the class[1]. Thus it is a class invariant. However in general it is not computable so that we need to perform an abstraction in order to safely approximate $\mathsf{c}[\![\mathtt{C}]\!]$.

Two observations on the above-defined class concrete semantics. The first one is that the domains D_{in}, $\mathsf{D}_{\mathsf{val}}$ and $\mathsf{D}_{\mathsf{out}}$ are left as a parameter, so that e.g. they can be instantiated to handle multiple parameters or to model memory and hence objects aliasing. The second observation is that (1) abstracts away from the values returned by a method. This is sound as far as the returned values do not expose the object internal state. If this is not the case, then the context can arbitrarily access and modify an object internal state. Hence, as a class invariant must hold for all the class objects in all the instantiation contexts then the only sound invariant for the exposed part of the object state is that "it can take any value". As a consequence, in such a case the problem of inferring a class invariant reduces to the use of an escape analysis [2] to determine which part of the object state is exposed to the context. In the rest of the paper we study the case in which the object state is encapsulated, so it can be modified only by the class methods.

4 Abstract Semantics

We use the abstract interpretation framework [5] in order to compute a safe upper-approximation of $\mathsf{c}[\![\mathtt{C}]\!]$. So, let $\langle \mathsf{D}^{\mathsf{a}}, \sqsubseteq^{\mathsf{a}} \rangle$ be an abstract domain approximating sets of states, i.e. it is linked to the concrete domain by a Galois connection:

$$\langle \wp(\Sigma), \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathsf{D}^{\mathsf{a}}, \sqsubseteq^{\mathsf{a}} \rangle.$$

Moreover let us consider the abstract counterparts for the constructor and the method collecting semantics such that the initial states are approximated by the abstract counterpart of the constructor semantics, i.e. $S_0 \subseteq \gamma(\mathsf{i}[\![\mathtt{init}]\!]^{\mathsf{a}})$ and the method semantics is soundly approximated by an abstract semantic function $\mathsf{M}[\![\mathtt{m}_i]\!]^{\mathsf{a}} \in [\mathsf{D}^{\mathsf{a}} \to \mathsf{D}^{\mathsf{a}}]$ such that $\forall \mathsf{d}^{\mathsf{a}} \in \mathsf{D}^{\mathsf{a}}.\ \alpha \circ \mathsf{M}[\![\mathtt{m}_i]\!] \circ \gamma(\mathsf{d}^{\mathsf{a}}) \sqsubseteq^{\mathsf{a}} \mathsf{M}[\![\mathtt{m}_i]\!]^{\mathsf{a}}(\mathsf{d}^{\mathsf{a}})$. Then it is possible to state the following theorem, that gives a characterization of class invariants as solutions of a system of recursive equations on D^{a} that mimics (1):

Theorem 1. *Let* $\mathtt{C} = \langle \mathtt{F}, \mathtt{init}, \{\mathtt{m}_1, \ldots \mathtt{m}_n\} \rangle$ *be a class,* D^{a} *an abstract domain such that* $\langle \wp(\Sigma), \subseteq, \cup \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathsf{D}^{\mathsf{a}}, \sqsubseteq^{\mathsf{a}}, \sqcup^{\mathsf{a}} \rangle$. *Moreover, let* $\mathsf{i}[\![\mathtt{init}]\!]^{\mathsf{a}} \in \mathsf{D}^{\mathsf{a}}$ *such that* $S_0 \subseteq \gamma(\mathsf{i}[\![\mathtt{init}]\!]^{\mathsf{a}})$ *and* $\mathsf{M}[\![\mathtt{m}_i]\!]^{\mathsf{a}} \in [\mathsf{D}^{\mathsf{a}} \to \mathsf{D}^{\mathsf{a}}]$ *be an abstract semantic function such that* $\alpha \circ \mathsf{M}[\![\mathtt{m}_i]\!] \circ \gamma \sqsubseteq^{\mathsf{a}} \mathsf{M}[\![\mathtt{m}_i]\!]^{\mathsf{a}}$. *Then* $I \in \mathsf{D}^{\mathsf{a}}$ *solution of the following recursive equation:*

$$I = \mathsf{i}[\![\mathtt{init}]\!]^{\mathsf{a}} \sqcup^{\mathsf{a}} \bigsqcup_{i=1}^{n\,\mathsf{a}} \mathsf{M}[\![\mathtt{m}_i]\!]^{\mathsf{a}}(I) \tag{2}$$

is such that $\mathsf{c}[\![\mathtt{C}]\!] \subseteq \gamma(I)$.

[1] In general, the body of a method $\mathtt{m}_i$ may invoke a method $\mathtt{m}_j$ that belongs to the same class. However, as such a call is somehow private to the class at this point the class invariant is not required to hold [14].

The least solution of the above system of equations can be computed using the standard Tarski-Kleene iterations. However, in general the height of the domain $\mathsf{D}^\mathtt{a}$ is infinite. In such a case, a widening operator $\nabla^\mathtt{a} \in [\mathsf{D}^\mathtt{a} \times \mathsf{D}^\mathtt{a} \to \mathsf{D}^\mathtt{a}]$ must be used to force the convergence of the iterations to a post-fixpoint [5].

The result of Th. 1 can be extended to cope with methods that expose (a part of) the internal state. For example, consider a method `m` that returns a reference to an object field `f`. Thus the context is free to arbitrary modify the value of the field `f`. This situation can be handled by considering an abstract domain $\mathsf{D}^\mathtt{a}$ expressive enough to contain escape information [2]. In such a setting, the static analysis of `m` will determine that `f` escapes its scope and the corresponding post-condition I_i will be such that[2] $I_i \downharpoonright \{\mathtt{f}\} = \top^\mathtt{a}$. This is the only sound assumption if we want to infer a class invariant valid for all the contexts.

Example 1. A class invariant for `Stack` can be inferred by instantiating $\mathsf{D}^\mathtt{a}$ with the Octagon abstract domain [15]:

$$I = \{1 \leq \mathtt{size}, 0 \leq \mathtt{pos} \leq \mathtt{size}, \mathtt{size} = \mathtt{stack.length}\}.$$

Modularity. Our approach is, at first *context*-modular as the derivation of a class invariant is done without any hypothesis on the module calling context. On the other side, our approach is not fully *has-a*-modular, i.e. if a class $\mathtt{C_i}$ to be analyzed has a variable (field, method formal parameter, etc.) of type $\mathtt{C_j}$ then $\mathtt{C_j}$ must be analyzed before $\mathtt{C_i}$. However the analysis of $\mathtt{C_i}$ does not strictly require the code of $\mathtt{C_j}$ and it can use the class invariant and the postconditions of the methods: any reference in $\mathtt{C_i}$ to an object of type $\mathtt{C_j}$ can be conservatively substituted by the $\mathtt{C_j}$ class invariant, and any invocation of a $\mathtt{C_j}$ method can be replaced by its postcondition. The advantage of this approach is that if two or more distinct classes use the same class $\mathtt{C_j}$ then it can be analyzed once and its result can be used many times, speeding up the whole analysis. Eventually, if two or more classes depend in a cyclic way, then they must either be analyzed together or the circularity must be broken with the technique of cutpoints described in [6, §8.3]. A third kind of modularity, *is-a*-modularity will be considered in the next section.

Full Program Analysis. In general an object-oriented program consists of a set of classes $\{\mathtt{C_i}\}$ and a main expression. For the moment we do not consider inheritance. From the set of classes we can statically derive the (inverse) graph of the *has-a* relation. This graph is such that the nodes are the classes of the program and there is an edge from $\mathtt{C_j}$ to $\mathtt{C_i}$ if and only if the class $\mathtt{C_i}$ has fields or local variables of type $\mathtt{C_j}$. The program analysis begins with the initial nodes, i.e. the nodes that have no predecessor. Once these nodes are analyzed (if possible in parallel on distinct computation units) the successors can be considered and so on. If a cycle is found, then the considerations of the last section apply. Most of the time, the analysis of a full program does not require the analysis of all the classes it uses. In fact, in general a program will use some library classes. These can be analyzed once (as our analysis is *context*-modular) and the result (re-) used by all the programs that use of them.

[2] $I \downharpoonright \mathtt{V}$ denotes the projection of the invariant I on the set of variables $\mathtt{V}$.

5 Class Invariant Inference in Presence of Inheritance

Now we address the problem of inferring a class invariant for a class $\mathtt{S}$ = "$\mathtt{E}$ `extends` $\mathtt{C}$", for some base class $\mathtt{C}$ and extension $\mathtt{E}$. The immediate method is to take the code of $\mathtt{C}$, that of $\mathtt{E}$ and then to apply the analysis described in the previous section to the expanded class $\mathtt{S}$ [16]. However such a naive approach has many drawbacks. The first one is code blow-up. In fact, it is known [16, §6.4] that the expansion of the inheritance causes (in the worst case) a quadratic blow up of the code size. Therefore, the direct analysis of the expanded code causes a quadratic loss of performances. Moreover, in general $\mathtt{C}$ can be the base class for two distinct extensions $\mathtt{E}$ and $\mathtt{E}'$. In that case the code $\mathtt{C}$ will be expanded and hence analyzed twice, with a further performances loss. Eventually, in some cases the $\mathtt{C}$ source code is not available, e.g. with applications that use third-party libraries. In that case, the library providers are unlikely to distribute the source code. A reasonable solution may be to ship the class invariants together with the compiled code. In that way the analysis of $\mathtt{S}$ will use the source code of $\mathtt{E}$ and the class invariant for $\mathtt{C}$, instead of its source. In order to do it, we consider the two orthogonal aspects of inheritance: class extension and method redefinition.

Class Extension. A subclass may extend the behavior of the superclass by adding methods. For the moment, we do not consider redefinition of methods, so that the general form of $\mathtt{S}$ is $\langle \mathtt{F_C} \cup \mathtt{F_E}, \mathtt{init}_E, \{\mathtt{m}_1 \dots \mathtt{m}_n\} \cup \{\mathtt{n}_1 \dots \mathtt{n}_k\}\rangle$, where $\mathtt{F_C}$ and $\mathtt{m}_i$ are the fields and the methods from the base class $\mathtt{C}$ and $\mathtt{F_E}$ and $\mathtt{n}_i$ those from the class extension $\mathtt{E}$. As a consequence, the equations system (2), instantiated for the subclass $\mathtt{S}$, becomes:

$$J = \mathsf{i}[\![\mathtt{init_E}]\!]^{\mathsf{a}} \sqcup^{\mathsf{a}} \bigsqcup_{i=1}^{n}{}^{\mathsf{a}} \mathsf{M}[\![\mathtt{m}_i]\!]^{\mathsf{a}}(J) \sqcup^{\mathsf{a}} \bigsqcup_{i=1}^{k}{}^{\mathsf{a}} \mathsf{M}[\![\mathtt{n}_i]\!]^{\mathsf{a}}(J). \tag{3}$$

Now, the goal is to solve the equation above in a smarter way than performing a brute fixpoint computation. In particular, we are interested in a solution that is function of the base class invariant, so that the methods $\mathtt{m}_i$ does not need to be analyzed again. In order to do it, the first remark is that the property $J_0 = \mathsf{i}[\![\mathtt{init_E}]\!]^{\mathsf{a}}$ is about the variables in $\mathtt{F_C} \cup \mathtt{F_E}$, so that it can be split in two parts $J_0 = J_0^{\mathtt{C}} \sqcup^{\mathsf{a}} J_0^{\mathtt{E}}$ where the first refers to the inherited fields and the latter to the $\mathtt{F_E}$. For what concerns $J_0^{\mathtt{C}}$ it is reasonable to assume $J_0^{\mathtt{C}} \sqsubseteq^{\mathsf{a}} I$, i.e. at creation time the subclass objects do not violate the superclass invariant [3]. This is a very common situation in object oriented programming: for example in C++ [7] it is a standard procedure to call the superclass constructors to set up the inherited fields and then to initialize the fields in $\mathtt{F_E}$ and even Java semantics [11] forces the initialization of the base class fields before the subclass constructor(s) can access them.

The next step is to look for a solution of (3) with a particular shape. Informally, $\mathtt{S}$ can behave either as the base class $\mathtt{C}$ or the extension $\mathtt{E}$. Thus it is reasonable to look for a solution in the form of $I \sqcup^{\mathsf{a}} X$, where I is the invariant of

[3] Recall that the order relation $\sqsubseteq^{\mathsf{a}}$ is the abstract counterpart of logical implication.

the base class $\mathtt{C}$ and X involves just the methods of $\mathtt{E}$. Formally, X is a solution of the following recursive equation:

$$X = J_0^{\mathtt{E}} \sqcup^{\mathsf{a}} \bigsqcup_{i=1}^{k\,\mathsf{a}} \mathsf{M}[\![\mathtt{n}_i]\!]^{\mathsf{a}}(I \sqcup^{\mathsf{a}} X). \tag{4}$$

As a consequence we obtain the following equation system:

$$\{(3), (4), J_0 = J_0^{\mathtt{C}} \sqcup^{\mathsf{a}} J_0^{\mathtt{E}}, J = I \sqcup^{\mathsf{a}} X\}. \tag{5}$$

It is worth noting that a solution J of (5) is a solution of (3), whereas in general the contrary does not hold. The interest of using (5) is that the computation of J reduces to the computation of (4), so that the subclass invariant J can be obtained using just the superclass invariant (as $J = I \sqcup^{\mathsf{a}} X$). Furthermore, it is possible to show that when analyzing a class hierarchy, in general the obtained speedup is linear in the number of direct descendants of a class and quadratic in the depth of the class hierarchy. The following theorem gives a sufficient and necessary condition for the existence of solutions of (3):

Theorem 2. *If* $\mathsf{M}[\![\cdot]\!]^{\mathsf{a}}$ *is a join-morphism then the equation system (5) has a solution iff*

$$\bigsqcup_{i=1}^{n\,\mathsf{a}} \mathsf{M}[\![\mathtt{m}_i]\!]^{\mathsf{a}}(X) \sqsubseteq^{\mathsf{a}} I \sqcup^{\mathsf{a}} X. \tag{6}$$

Proof. Using the identities of (5) it is possible to rewrite the equation (3) as follows:

$$\begin{aligned}
J =& \mathsf{i}[\![\mathtt{init}_{\mathtt{E}}]\!]^{\mathsf{a}} \sqcup^{\mathsf{a}} \bigsqcup_{i=1}^{n\,\mathsf{a}} \mathsf{M}[\![\mathtt{m}_i]\!]^{\mathsf{a}}(J) \sqcup^{\mathsf{a}} \bigsqcup_{i=1}^{k\,\mathsf{a}} \mathsf{M}[\![\mathtt{n}_i]\!]^{\mathsf{a}}(J) \\
=& J_0^{\mathtt{C}} \sqcup^{\mathsf{a}} J_0^{\mathtt{E}} \sqcup^{\mathsf{a}} \bigsqcup_{i=1}^{n\,\mathsf{a}} \mathsf{M}[\![\mathtt{m}_i]\!]^{\mathsf{a}}(I \sqcup^{\mathsf{a}} X) \sqcup^{\mathsf{a}} \bigsqcup_{i=1}^{k\,\mathsf{a}} \mathsf{M}[\![\mathtt{n}_i]\!]^{\mathsf{a}}(I \sqcup^{\mathsf{a}} X) \\
=& J_0^{\mathtt{C}} \sqcup^{\mathsf{a}} \bigsqcup_{i=1}^{n\,\mathsf{a}} \mathsf{M}[\![\mathtt{m}_i]\!]^{\mathsf{a}}(I) \sqcup^{\mathsf{a}} \bigsqcup_{i=1}^{n\,\mathsf{a}} \sqcup^{\mathsf{a}} \mathsf{M}[\![\mathtt{m}_i]\!]^{\mathsf{a}}(X) \sqcup^{\mathsf{a}} J_0^{\mathtt{E}} \sqcup^{\mathsf{a}} \bigsqcup_{i=1}^{k\,\mathsf{a}} \mathsf{M}[\![\mathtt{n}_i]\!]^{\mathsf{a}}(I \sqcup^{\mathsf{a}} X) \\
=& I \sqcup^{\mathsf{a}} \bigsqcup_{i=1}^{n\,\mathsf{a}} \mathsf{M}[\![\mathtt{m}_i]\!]^{\mathsf{a}}(X) \sqcup^{\mathsf{a}} X
\end{aligned}$$

From basic lattice theory, it follows that the above equation is consistent with $J = I \sqcup^{\mathsf{a}} X$ iff (2) holds. □

If the subclass preserves the parent invariant w.r.t. the inherited fields, i.e. $X \downharpoonright \mathtt{F_C} \sqsubseteq^{\mathsf{a}} I$ then the above equation system admits solutions. In general it may be difficult to prove the theorem hypothesis, and in the worst case it is computationally equivalent to solve (3). However, it can be shown that a large class of static analyses, namely the *symbolic relational* ones [13,6,12] satisfies the hypothesis of the theorem, so that (2) must be checked once and for all.

Examples of symbolic relational static analyses are Octagons, Relevant Context Inference [4] and types. Next example shows that whilst a solution of (5) is a solution of (3) in general it is not the *least* solution.

Example 2. The class `ClosedSystem` which models closed physical systems with a total energy $\mathtt{c}_0$ and two different kinds of internal energy `a` and `b`, can be defined as $\mathtt{C} = \langle \{\mathtt{a}, \mathtt{b}\}, \mathtt{init}, \{\mathtt{add}, \mathtt{sub}\}\rangle$. The constructor is $\mathtt{init} = \lambda(\mathtt{a}_0, \mathtt{c}_0).\ (\mathtt{a} := \mathtt{a}_0;\ \mathtt{b} := \mathtt{c}_0 - \mathtt{a}_0)$ and the two methods are: $\mathtt{add} = \lambda().\ (\mathtt{a} := \mathtt{a} + 1;\ \mathtt{b} := \mathtt{b} - 1)$ and $\mathtt{sub} = \lambda().\ (\mathtt{a} := \mathtt{a} - 1;\ \mathtt{b} := \mathtt{b} + 1)$. Using (2) and the Octagon abstract domain it is possible to infer the class invariant $I = \{\mathtt{a} + \mathtt{b} = \mathtt{c}_0\}$. Let us consider the extension `ExtendedSystem` with a field `c`, a new constructor $\mathtt{init} = \lambda(\mathtt{a}_0, \mathtt{c}_0).\ (\mathtt{super.init}(\mathtt{a}_0,\ \mathtt{c}_0);\ \mathtt{c} := \mathtt{c}_0)$ and a method $\mathtt{ext} = \lambda().\ (\mathtt{c} := \mathtt{c} - 1;\ \mathtt{a} := \mathtt{a} - 1)$.

If we apply (5) we obtain $X = \{\mathtt{a} + \mathtt{b} = \mathtt{c},\ \mathtt{c} \leq \mathtt{c}_0\}$ and $J = I \sqcup^{\mathtt{a}} X = \{\mathtt{a} + \mathtt{b} \leq \mathtt{c}_0,\ \mathtt{c} \leq \mathtt{c}_0\}$. However, the direct application of (3) gives $J' = \{\mathtt{a} + \mathtt{b} = \mathtt{c},\ \mathtt{c} \leq \mathtt{c}_0\}$. It is immediate to see that J' is a more precise invariant than J, that is $J' \sqsubseteq^{\mathtt{a}} J$. □

Methods Redefinition. A subclass may redefine the behavior of the superclass by redefining some of its methods. The modalities for doing it largely depend on the considered object oriented language. For example C++, C# and Java apply syntax criteria (the overriding method must have the same name and type as the overridden) whereas in Eiffel [14] a method `n` overrides `m` if and only if the type of `n` is (co-variantly) a subtype of `m`. Nevertheless in order to be as language-independent as possible, we consider just how overriding and overridden methods interact and not how the overriding is provided by the language. However an implementation of an analyzer for a real language must consider this point.

As usual in abstract interpretation, we proceed by successive approximations. First we assume that all the methods of `S` may be executed, even those that are redefined, so that the results of the previous section apply directly. This is an over-approximation of the real behavior: if a method is redefined in a subclass, then it is not directly accessible by the context. Nevertheless, the redefined method is still reachable by the method bodies of `E`. In general, because of late-binding, we must distinguish two situations: *down*calls and *up*calls: in the first case a method of the parent class invokes one that has been redefined and in the latter the interaction happens in the opposite direction. Once again, we can handle both by upper-approximating their behavior. Let us consider a method definition in the form $\mathtt{m} = \lambda\mathtt{x}_{\mathtt{in}}.\ (\ldots;\ \mathtt{v} := \mathtt{m}_{call}(\mathtt{y}); \ldots)$, for some variables `v` and `y`.

If the invocation of $\mathtt{m}_{call}$ may resolve in a downcall, then a safe (but rather imprecise) approximation is to consider that an overriding method can arbitrarily modify the object internal state. Then the object state at the program point just after the assignment can be any $\sigma \in \Sigma$. Therefore when performing the analysis of the `m` body, the abstract environment just after the assignment will be set equal to $\alpha(\Sigma) = \top^{\mathtt{a}}$, the largest element of the abstract domain. An improvement would be to use I instead of $\alpha(\Sigma)$, but then the subclass must

be checked to preserve the invariant I, i.e. $J \lfloor \mathtt{F_C} \sqsubseteq^{\mathsf{a}} I$. A better solution for down-calls in the general case is subject of future work.

On the other hand, if $\mathtt{m}_{call}$ resolves in an upcall then a worst case approximation of $\mathsf{M}[\![\mathtt{m}_{call}]\!]$ must be employed so that $\mathtt{v} := \mathsf{M}[\![\mathtt{m}_{call}]\!]^{\mathsf{a}}(\top^{\mathsf{a}})$. Why this? Essentially because in the `m` body the (super)class invariant may not hold [14] so that it is not sound to assume it as an approximation of $\mathtt{m}_{call}$ semantics. Therefore, if we want to analyze the subclass code without referring to the parent one then the only sound assumption is to consider the $\mathtt{m}_{call}$ postcondition when its input is not known, i.e. it can be everything. In this last case, the base class must be shipped not only with the invariant I but also with an approximation $\mathsf{M}[\![\mathtt{m}_i]\!]^{\mathsf{a}}(\top^{\mathsf{a}})$ for each method $\mathtt{m}_i$.

Example 3. A class invariant for `StackWithUndo` can be inferred applying the considerations above and instancing the equation system (5). It is easy to see that the constructor preserves the `Stack` invariant, so $J_0 = I \sqcup^{\mathsf{a}} \{\mathtt{undoType} = 0\}$.

Then we can consider the application of (4) to `undo` and to the overridden methods `push` and `pop`. The three perform an upcall. In that case it is sound to *replace* it with the `Stack` invariant as it holds at the program point just before the `super.pop` and `super.push` invocations. Therefore it is immediate to obtain $X = I \sqcup^{\mathsf{a}} \{-1 \leq \mathtt{undoType} \leq 1\} = I \sqcup^{\mathsf{a}} X = J$, the class invariant for `StackWithUndo`.

If it is needed, the obtained invariant and method post-conditions can be improved by using the incremental refinement technique of [10]. The abstract domain can be refined by partitioning it through the values assumed by `undoType` (that using the already computed class invariant J are at most 3). In such a way it is possible to infer the properties: $\mathtt{undoType} = 1 \Rightarrow \mathtt{pos} > 0$ and $\mathtt{undoType} = 0 \Rightarrow 0 \leq \mathtt{pos} \leq \mathtt{size}$ and $\mathtt{undoType} = -1 \Rightarrow \mathtt{pos} < \mathtt{size}$ so that it is proved that `undo` never throws the exception `StackError`. □

6 Conclusions and Future Work

In this work we presented a framework for class modular analysis of object oriented languages in presence of inheritance. We derived the equations for the class invariant and we instantiated them in the case of inheritance. We discussed the resolvability whenever a solution, function of the base class invariant, is required. Eventually, we showed how to handle up-calls and down-calls.

We have a prototype implementation of the framework presented in this paper with a partial support for inheritance. In particular, the fixpoint computation is implemented by an OCAML functor, whom parameter is an abstract domain for the approximation of the environments (cfr. Th. 1). From our preliminary tests it seems that relational domains are best-suited for effective and precise analyses. In fact, such domains allow to keep a relation between the constructor parameters and the object internal state. For instance in the `Stack` example the relation between the `Stack` constructor parameter, i.e. the stack size, and the stack pointer is inferred. On the other hand, if the analyzer is instantiated with the interval domain [5] the inferred class invariant is $\{\mathtt{pos} \in [0, +\infty]\}$, that is too rough to be practical. We plan to extend our prototype implementation in

order to practical study its effectiveness and in particular the handling of the inheritance.

Acknowledgments. We thank B. Blanchet, P. and R. Cousot, J. Feret, R. Giacobazzi, C. Hymans, A. Miné, X. Rival and D. A. Schmidt for the stimulating discussions and the comments on this paper.

References

1. A. Aggarwal and K. H. Randall. Related field analysis. In *PLDI '01*. ACM Press, June 2001.
2. B. Blanchet. Escape analysis for object oriented languages. Application to Java. In *OOPSLA'99*. ACM Press, November 1999.
3. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Min, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety-Critical Software. In *PLDI'03*. ACM Press, June 2003.
4. R. Chatterjee, B. G. Ryder, and W. A. Landi. Relevant context inference. In *POPL '99*. ACM Press, January 1999.
5. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*. ACM Press, January 1977.
6. P. Cousot and R. Cousot. Modular static program analysis, invited paper. In *CC'02*, volume 2304 of *LNCS*. Springer-Verlag, April 2002.
7. B. Eckel. *Thinking in C++, 2nd Edition*, volume 1. Prentice Hall, 2000.
8. M. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington Department of Computer Science and Engineering, 2002.
9. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI'02*. ACM Press, June 2002.
10. S. Genaim and M. Codish. Incremental refinement of semantic based program analysis for logic programs. In *ACSC'99*, volume 1587 of *LNCS*. Springer-Verlag, January 1999.
11. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification – Second Edition*. Sun Microsystems, 2001.
12. F. Logozzo. Class-level modular analysis for object oriented languages. In *SAS'03*, volume 2694 of *LNCS*. Springer-Verlag, June 2003.
13. F. Logozzo. Approximating module semantics with constraints. In *SAC 2004*. ACM Press, March 2004.
14. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.
15. A. Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*. IEEE CS Press, October 2001.
16. J. Palsberg and M.I. Schwartzbach. *Object-Oriented Type Systems*. John Wiley & Sons, Chichester, 1994.
17. G. Ramalingam, A. Warshavsky, J. Field, D. Goyal, and M. Sagiv. Deriving specialized program analyses for certifying component-client conformance. In *PLDI'02*, June 2002.

Liveness with Invisible Ranking*

Yi Fang[1], Nir Piterman[2], Amir Pnueli[1,2], and Lenore Zuck[1]

[1] New York University, New York, {yifang,zuck}@cs.nyu.edu
[2] Weizmann Institute of Science, Rehovot, Israel {nirp,amir}@wisdom.weizmann.ac.il

Abstract. The method of Invisible Invariants was developed originally in order to verify safety properties of parameterized systems fully automatically. Roughly speaking, the method is based on a *small model property* that implies it is sufficient to prove some properties on small instantiations of the system, and on a heuristic that generates candidate invariants. Liveness properties usually require well founded ranking, and do not fall within the scope of the small model theorem. In this paper we develop novel proof rules for liveness properties, all of whose proof obligations are of the correct form to be handled by the small model theorem. We then develop abstraction and generalization techniques that allow for fully automatic verification of liveness properties of parameterized systems. We demonstrate the application of the method on several examples.

1 Introduction

Uniform verification of parameterized systems is one of the most challenging problems in verification today. Given a parameterized system $S(N) : P[1] \| \cdots \| P[N]$ and a property p, uniform verification attempts to verify $S(N) \models p$ for every $N > 1$. One of the most powerful approaches to verification which is not restricted to finite-state systems is *deductive verification*. This approach is based on a set of proof rules in which the user has to establish the validity of a list of premises in order to validate a given property of the system. The two tasks that the user has to perform are:

1. Identify some auxiliary constructs which appear in the premises of the rule.
2. Establish the logical validity of the premises, using the auxiliary constructs identified in step 1.

When performing manual deductive verification, the first task is usually the more difficult, requiring ingenuity, expertise, and a good understanding of the behavior of the program and the techniques for formalizing these insights. The second task is often performed using theorem provers such as PVS [22] or STeP [4], which also require extensive user expertise and ingenuity. The difficulty in the execution of these two steps is the main reason why deductive verification is not used more extensively.

A representative case is the verification of invariance properties using the *invariance rule* of [18]. In order to prove that assertion r is an invariant of program P, the rule requires coming up with an auxiliary assertion φ which is *inductive* (i.e. is implied by the

* This research was supported in part by the Minerva Center for Verification of Reactive Systems, the European Community IST project "Advance", the Israel Science Foundation (grant no. 106/02-1), and NSF grant CCR-0205571.

B. Steffen and G. Levi (Eds.): VMCAI 2004, LNCS 2937, pp. 223–238, 2004.

initial condition and is preserved under every computation step) and which strengthens (implies) r.

In [20,2] we introduced the method of *invisible invariants*, which proposes a method for automatic generation of the auxiliary assertion φ for parameterized systems, as well as an efficient algorithm for checking the validity of the premises of the invariance rule. In this paper we generalize the method of invisible invariants to deal with liveness properties.

The method of invisible invariants is based on two main ideas:

It is often the case that the auxiliary assertion for a parameterized system has the form $\varphi : \forall i.q(i)$ (or, more generally, $\forall i \neq j.q(i,j)$.) We construct an instance of the parameterized system taking a fixed value N_0 for the parameter N. For the finite-state system $S(N_0)$, we compute the set of reachable states $reach$ using a symbolic model checker. Let r_1 be the projection of $reach$ on process index 1, obtained by discarding references to all variables which are local to all processes other than $P[1]$. We take $q(i)$ to be the generalization of r_1 obtained by replacing each reference to a local variable $P[1].x$ by a reference to $P[i].x$. The obtained $q(i)$ is our candidate for the body of the inductive assertion $\varphi : \forall i.q(i)$. We refer to this part of the process as *project-and-generalize*.

Having obtained a candidate for the inductive assertion φ, we still have to check the validity of the premises of the invariance rule. Under the assumption that our assertional language is restricted to the predicates of equality and inequality between bounded range integer variables (which is adequate for many of the parameterized systems we considered), and the fact that our candidate inductive assertions have the form $\forall \vec{i}.q(\vec{i})$, we managed to prove a *small model* theorem. According to this theorem, there exists a (small) bound N_0 such that the premises of the invariance rule are valid for every N iff they are valid for all $N \leq N_0$. This enables us to use BDD techniques in order to check the validity of the premises. This theorem is based on the fact that, under the above assumptions, all premises can be written in the form $\forall \vec{i} \exists \vec{j}.\psi(\vec{i},\vec{j})$, where $\psi(\vec{i},\vec{j})$ is a quantifier-free assertion that may only refer to the global variables and the local variables of $P[i]$ and $P[j]$.

Being able to validate the premises on $S[N_0]$ has the additional important advantage that the user does not have to see the automatically generated auxiliary assertion φ. This assertion is generated as part of the procedure and is immediately used in order to validate the premises of the rule. This is advantageous because, being generated by symbolic BDD techniques, its representation is often extremely unreadable and non-intuitive, and will usually not contribute to a better understanding of the program or its proof. Because the user never gets to see the auxiliary invariant, we refer to this method as the method of *invisible invariants*.

In this paper, we extend the method of invisible invariants to apply to proofs of the second most important class of properties which the class of *response properties*. These are liveness properties which can be specified by the temporal formula $\Box(q \rightarrow \Diamond r)$ (also written as $q \Rightarrow \Diamond r$) and guarantees that any q-state is eventually followed by a r-state. To do so, we consider a certain variant of rule WELL [17] which establishes the validity of response properties under the assumption of *justice* (weak fairness). As is well known to users of this and similar rules, such a proof requires the generation of two kinds of auxiliary constructs: "helpful" assertions which characterize the states at which a certain

transition is helpful in promoting progress towards the goal (r), and *ranking functions* which measure the progress towards the goal.

In order to make the *project-and-generalize* technique applicable to the automatic generation of the ranking functions, we developed a special variant of rule WELL, to which we refer as rule DISTRANK. In this version of the rule, we associate with each potentially helpful transition τ_i an individual ranking function $\delta_i : \Sigma \mapsto [0..c]$, mapping states to integers in a small range $[0..c]$, where c is a fixed small constant, independent of the parameter N. The global ranking function can be obtained by forming the multi-set $\{\delta_i\}$. In most of the examples we considered, it was sufficient to take the range of the individual functions to be $\{0, 1\}$, which enable us to view each δ_i as an assertion, and generate it automatically using the *project-and-generalize* technique.

The paper is organized as follows: In Section 2, we present the general computational model of FDS and the restrictions which enable to (invisibly) obtain auxiliary constructs. In this section we also review the small model property which enables to automatically validate the premises of the various proof rules.

In Section 3 we introduce the new DISTRANK proof rule, explain how we automatically generate ranking and helpful assertions for the parameterized case, and demonstrate the techniques on the two examples of a TOKEN-RING and the BAKERY algorithms. The method introduced in this section is adequate for all the cases in which the set of reachable states can be satisfactorily over-approximated by an assertion of the form $\forall i.u(i)$ and both the helpful assertion h_i and the individual ranking function δ_i can be represented by an assertion of the form $\alpha(i)$ which only refers to the local variables of process $P[i]$ and to the global variables.

Not all examples can be handled by assertions which depend on a single parameter. In Section 4, we consider cases whose verification requires a characterization of the reachable states by an assertion of the form $\forall i.u(i) \ \wedge \ \exists j.e(j)$. In a future paper we will consider helpful assertions h_i which have the form $h_i : \forall j \neq i.\psi(i, j)$. With these extensions we can handle another version of the BAKERY algorithm. A variant of this method has been successfully applied to Szymanski's N-process mutual exclusion algorithm.

Related Work. In [21] we introduced the method of "counter-abstraction" to automatically prove liveness properties of parameterized systems. Counter-abstraction is an instance of data-abstraction [13] and has proven successful in instances of systems with a trivial (star or clique) topologies and a small state-space for each process. The work there is similar to the work of the PAX group (see, e.g., [3]) which is based on the method of *predicate abstraction* [5]. While there are several differences between the two approaches, both are not "fully automatic" in the sense that the user has to provide the system with abstraction methodology.

In [23,14] we used the method of *network invariants* [13] to prove liveness properties of parameterized systems. While extremely powerful, the main weakness of the method is that the abstraction is performed manually by the user.

The problem of uniform verification of parameterized systems is, in general, undecidable [1]. One approach to remedy this situation, pursued, e.g., in [8], is to look for restricted families of parameterized systems for which the problem becomes decidable.

Many of these approaches fail when applied to asynchronous systems where processes communicate by shared variables.

Another approach is to look for sound but incomplete methods. Representative works of this approach include methods based on: explicit induction ([9]), network invariants that can be viewed as implicit induction ([16]), abstraction and approximation of network invariants ([6]), and other methods based on abstraction ([10]). Other methods include those relying on "regular model-checking" (e.g., [12]) that are often specially effective due to the application of *acceleration* procedures but are not applciable to all cases, methods based on symmetry reduction (e.g., [11]), or compositional methods (e.g., ([19]) that combine automatic abstraction with finite-instantiation due to symmetry. Some of these works, from which we have mentioned only few representatives, require the user to provide auxiliary constructs and thus do not provide for fully automatic verification of parameterized systems. Others (such as regular model checking) are applicable to restricted classes of parameterized systems, as is our method. Almost none of these methods have been applied to the verification of liveness properties, which we address in this paper.

Less related to our work is the work in [7] which presents methods for obtaining ranking functions for sequential programs. Another interesting paper which deals with program temination is [15]. However, the treatment there is also restricted to sequential program and does not address parameterized systems.

2 The Model

As our basic computational model, we take a *fair discrete system* (FDS) $S = \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$, where

- V — A set of *system variables*. A *state* of the system S provides a type-consistent interpretation of the system variables V. For a state s and a system variable $v \in V$, we denote by $s[v]$ the value assigned to v by the state s. Let Σ denote the set of all states over V.
- Θ — The *initial condition*: An assertion (state formula) characterizing the initial states.
- $\rho(V, V')$ — The *transition relation*: An assertion, relating the values V of the variables in state $s \in \Sigma$ to the values V' in an S-successor state $s' \in \Sigma$.
- $\mathcal{J}$ — A set of *justice* (*weak fairness*) requirements: Each justice requirement is an assertion; A computation must include infinitely many states satisfying the requirement.
- $\mathcal{C}$ — A set of *compassion (strong fairness)* requirements: Each compassion requirement is a pair $\langle p, q \rangle$ of state assertions; A computation should include either only finitely many p-states, or infinitely many q-states.

A *computation* of an FDS S is an infinite sequence of states $\sigma : s_0, s_1, s_2, ...$, satisfying the requirements:

- *Initiality* — s_0 is initial, i.e., $s_0 \models \Theta$.
- *Consecution* — For each $\ell = 0, 1, ...$, the state $s_{\ell+1}$ is a S-successor of s_ℓ. That is, $\langle s_\ell, s_{\ell+1} \rangle \models \rho(V, V')$ where, for each $v \in V$, we interpret v as $s_\ell[v]$ and v' as $s_{\ell+1}[v]$.

- *Justice* — for every $J \in \mathcal{J}$, σ contains infinitely many occurrences of J-states.
- *Compassion* – for every $\langle p, q \rangle \in \mathcal{C}$, either σ contains only finitely many occurrences of p-states, or σ contains infinitely many occurrences of q-states.

For simplicity, all of our examples will contain no compassion requirements, i.e., $\mathcal{C} = \emptyset$. Most of the methods can be generalized to deal with systems with a non-empty set of compassion requirements.

2.1 Fair Bounded Discrete Systems

To allow the application of the invisible constructs methods, we place further restrictions on the systems we study, leading to the model of *fair bounded discrete systems* (FBDS), that is essentially the model of bounded discrete systems of [2] augmented with fairness. For brevity, we describe here a simplified three-type model; the extension for the general multi-type case is straightforward.

Let $N \in \mathbb{N}^+$ be the *system's parameter*. We allow the following data types:

1. **bool**: the set of boolean and finite-range scalars;
2. **index**: a scalar type that includes integers in the range $[1..N]$. These serve as indices to arrays and processes;
3. **par_data**: a scalar data type that includes integers in the range $[0..N]$; and
4. Arrays of the type **index** $\mapsto$ **bool** and **index** $\mapsto$ **par_data**.

For simplicity, we allow the system variables to include any number of variables of types **bool**, **index**, and **index** $\mapsto$ **bool**, but at most a single variable of type **index** $\mapsto$ **par_data**, and no variables of type **par_data**.

Atomic formulas may compare two variables of the same type. E.g., if y and y' are **index** variables, and z is a **index** $\mapsto$ **par_data**, then $y = y'$ and $z[y] < z[y']$ are both atomic formulas. We admit $0, 1$ as constants of type **bool**, $1, N$ as constants of type **index**, and $0, N$ as constants of type **par_data**. Apparent conflicts arising out of the overloading of the constant symbols can always be resolved by type analysis. We refer to formulas obtained by boolean combinations of such atomic formulas as *restricted boolean assertions* (or *boolean assertions* for short). Applying quantification over **index** variables to boolean assertions, we obtain the class of *restricted assertions*.

As the initial condition Θ, we allow restricted assertions of the form $\forall i.u(i) \wedge \exists j.e(j)$, where $u(i)$ and $e(j)$ are boolean assertions.

As the transition relation ρ, we allow restricted assertions of the form $\exists \vec{i} \forall \vec{j}.\psi(\vec{i}, \vec{j})$ for a boolean assertion $\psi(\vec{i}, \vec{j})$.

The allowed justice requirements are boolean assertions which may be parameterized by an **index** variable.

The reason we distinguish between the type **index** and **par_data**, even though they range over similar data domains, is to avoid subscript expressions of the form $a[b[i]]$, which will invalidate many of the methods developed in the sequel.

Example 1 (The Token Ring Algorithm).
Consider program TOKEN-RING in Fig. 1, which is a mutual exclusion algorithm for any N processes.

$$
\begin{array}{l}
\textbf{in}\ \ N\ : \textbf{natural where}\ N > 1 \\
\quad\ \ \mathit{tloc} : [1..N] \\
\left\|_{i=1}^{N} P[i] :: \right. \left[\begin{array}{l}
\textbf{loop forever do} \\
\left[\begin{array}{l}
\ell_0 : \textbf{if}\ \mathit{tloc} = i\ \textbf{then}\ \mathit{tloc} := \mathit{tloc} \oplus_N \mathbf{1} \\
\qquad \textbf{go to}\ \{\ell_0, \ell_1\} \\
\ell_1 : \textbf{await}\ \mathit{tloc} = i \\
\ell_2 : \textbf{Critical}
\end{array}\right]
\end{array}\right]
\end{array}
$$

Fig. 1. Program TOKEN-RING

In this version of the algorithm, the global variable *tloc* represents the current location of the token. Location ℓ_0 constitutes the non-critical section which may nondeterministically exit to the trying section at location ℓ_1. While being in the non-critical section, a process guarantees to move the token to its right neighbor, whenever it receives it. This is done by incrementing *tloc* by 1, modulo N. At the trying section, a process $P[i]$ waits until it received the token which is signaled by the condition $\mathit{tloc} = i$.

Following is the FBDS corresponding to program TOKEN-RING:

$$
\begin{array}{l}
V : \left\{\begin{array}{l} \mathit{tloc} : [1..N] \\ \pi : \quad \textbf{array}[1..N]\ \textbf{of}\ [0..2] \end{array}\right. \\
\Theta : \forall i.\pi[i] = 0 \\
\rho : \ \exists i : \forall j \neq i : (\pi'[j] = \pi[j]) \wedge \\
\qquad \left[\begin{array}{ll}
 & \pi[i] = 0 \ \wedge\ \mathit{tloc} = i \ \wedge\ \mathit{tloc}' = i \oplus_N 1 \ \wedge\ \pi'[i] \in \{0, 1\} \\
\vee & \mathit{tloc}' = \mathit{tloc} \ \wedge\ \left(\begin{array}{l} \pi'[i] = \pi[i] \\ \vee\ \pi[i] = 0 \ \wedge\ \mathit{tloc} \neq i \ \wedge\ \pi'[i] = 1 \\ \vee\ \pi[i] = 1 \ \wedge\ \mathit{tloc} = i \ \wedge\ \pi'[i] = 2 \\ \vee\ \pi[i] = 2 \ \wedge\ \pi'[i] = 3 \end{array}\right)
\end{array}\right] \\
\mathcal{J} : \left(\begin{array}{ll}
\{J_0[i] : \neg(\pi[i] = 0 \ \wedge\ \mathit{tloc} = i) & \mid i \in [1..N]\} \quad \cup \\
\{J_1[i] : \neg(\pi[i] = 1 \ \wedge\ \mathit{tloc} = i) & \mid i \in [1..N]\} \quad \cup \\
\{J_2[i] : \quad \pi[i] \neq 2 & \mid i \in [1..N]\}
\end{array}\right)
\end{array}
$$

Note that *tloc* is a variable of type **index**, while the program counter π is a variable of type **index** $\mapsto$ **bool**.

Strictly speaking, the transition relation as presented above does not fully conform to the definition of a boolean assertion since it contains the atomic formula $\mathit{tloc}' = i \oplus_N 1$. However, this can be rectified by a two-stage reduction. First, we replace $\mathit{tloc}' = i \oplus_N 1$ by $(i < N \ \wedge\ \mathit{tloc}' = i + 1) \ \vee\ (i = N \ \wedge\ \mathit{tloc}' = 1)$. Then, we replace the formula $\exists i : \forall j \neq i : (\ldots \mathit{tloc}' = i + 1 \ldots)$ by
$\exists i, i_1 : \forall j \neq i, j_1 : (j_1 \leq i \ \vee\ i_1 \leq j_1) \ \wedge\ (\ldots \mathit{tloc}' = i_1 \ldots)$ which guarantees that $i_1 = i + 1$.

Let α be an assertion over V, and R be an assertion over $V \cup V'$, which can be viewed as a transition relation. We denote by $\alpha \circ R$ the assertion characterizing all state which are R-successors of α-states. We denote by $\alpha \circ R^*$ the states reachable by an R-path of length zero or more from an α-state.

2.2 The Small Model Theorem

Let $\varphi : \forall\vec{i}\exists\vec{j}.R(\vec{i},\vec{j})$ be an AE-formula, where $R(\vec{i},\vec{j})$ is a boolean assertion which refers to the state variables of a parameterized FBDS $S(N)$ and to the quantified variables $\vec{i}$ and $\vec{j}$. Let N_0 be the number of universally quantified and free **index** variables and constants appearing in R. We always include in the count N_0 the **index** constants 1 and N, even if they do not appear explicitly in R. The following claim (stated first in [20] and extended in [2]) provides the basis for the automatic validation of the premises in the proof rules:

Claim (Small model property).
Formula φ is valid iff it is valid over all instances $S(N)$ for $N \leq N_0$.

Proof Sketch
To prove the claim, we consider the negation of the formula, given by $\psi : \exists\vec{i}\forall\vec{j}.\neg R(\vec{i},\vec{j})$, and show that if ψ is satisfiable, then it is satisfiable in a state of a system $S(N)$ for some $N \leq N_0$. Assume that ψ is satisfiable in state s_{N_1} of an instance $S(N_1)$. Let $1 = u_1 < u_2 < \cdots < u_k = N$ be the sequence of pairwise distinct values of **index** variables and constants which appear free or existentially quantified within ψ, sorted in ascending order. Obviously $k \leq N_0$. We construct a state s_k in the system $S(k)$ as follows. Each **index** variable or constant that was interpreted as u_i in s_{N_1} is reinterpreted as i in s_k. For each array a : **index** $\mapsto$ **bool** and each $i = 1, \ldots, k$, we reinterpret $s_k[a[i]]$ as $s_{N_1}[a[u_i]]$. In case the system contains a (single) array b : **index** $\mapsto$ **par_data**, we also perform the following reduction: Let S be the set of all values which appear as the interpretation of $b[u_1], \ldots, b[u_k]$ in s_{N_1}, to which we add the constants 0 and $N = N_1$. Let $0 = v_0 < v_1 < \cdots < v_m = N_1$ be a sorted list of the values appearing in S. Obviously, $m \leq k$. We reinterpret the values of $b[i]$ in s_k as follows. If $s_{N_1}[b[u_i]] = v_j < N_1$, we let $s_k[b[i]] = j$. If $s_{N_1}[b[u_i]] = N_1$, we let $s_k[b[i]] = k$.

With the understanding that arrays in the system represent the local variables of individual processes, the reduction from a state of system $S(N_1)$ to a state of system $S(k)$, amounts to the preservation of processes $P[u_1], \ldots, P[u_k]$, renaming their indices to $P[1], \ldots, P[k]$, and discarding any process $P[j]$ in S_{N_1} such that $j \notin \{u_1, \ldots, u_k\}$. Discarding these processes preserve the validity of ψ because the reference to these "other" processes is only through a universal quantification $\forall\vec{j}$. So narrowing the set over which we universally quantify can only make the property "truer". This explains why the method is restricted to AE-formulas.

It follows that state s_k satisfies ψ. □

In case the formula φ refers to an expression such as $i+1$, we add 1 more variable which participates in the re-expression of $i+1$.

3 The Method of Invisible Ranking

In this section we present a new proof rule for liveness properties of an FBDS that allows, in some cases, to obtain an automatic verification of liveness properties for systems of any size. We first describe the new proof rule. We then present methods for the automatic generation of the auxiliary constructs required by the rule. We illustrate the application of these method on the example of TOKEN-RING as we go along.

3.1 A Distributed Ranking Proof Rule

In Fig. 2, We present proof rule DISTRANK (for *distributed ranking*), for verifying response properties. This rule assumes that the system to be verified has only justice requirements (in particular, it has no compassion requirements.)

For a parameterized system with a transition domain $\mathcal{T}(N)$
set of states $\Sigma(N)$,
justice requirements $\{J_\tau \mid \tau \in \mathcal{T}\}$,
invariant assertion φ,
assertions $q, r, pend$ and $\{h_\tau \mid \tau \in \mathcal{T}\}$,
and ranking functions $\{\delta_\tau : \Sigma \to \{0,1\} \mid \tau \in \mathcal{T}\}$

D1. $q \wedge \varphi \rightarrow r \vee pend$
D2. $pend \wedge \rho \rightarrow r' \vee pend'$
D3. $pend \rightarrow \bigvee_{\tau \in \mathcal{T}} h_\tau$
D4. $pend \wedge \rho \rightarrow r' \vee \bigwedge_{\tau \in \mathcal{T}} \delta_\tau \geq \delta'_\tau$
For every $\tau \in \mathcal{T}$
D5. $h_\tau \wedge \rho \rightarrow r' \vee h'_\tau \vee \delta_\tau > \delta'_\tau$
D6. $h_\tau \rightarrow \neg J_\tau$

$$q \Rightarrow \diamondsuit r$$

Fig. 2. The liveness rule DISTRANK

The rule is configured to deal directly with parameterized systems. As in other rules for verifying response properties (e.g.,[17]), progress is accomplished by the actions of *helpful transitions* in the system. In a parameterized system, the set of transitions has the structure $\mathcal{T}(N) = [0..m] \times N$ for some fixed m. Typically, $[0..m]$ enumerates the locations within process. For example, in program TOKEN-RING, $\mathcal{T}(N) = [0..2] \times N$, where each transition $\tau_k[i]$ is associated with location $m \in [0..2]$ within process $i \in [1..N]$. Each transition $\tau_k[i]$ is affiliated with a justice requirement $J_m[i]$, asserting that transition $\tau_k[i]$ is disabled. By requiring that $J_k[i]$ holds infinitely many times, we guarantee that $\tau_k[i]$ will be disabled infinitely many times, thus $\tau_k[i]$ will not be continuously enabled without being taken.

Assertion φ is an invariant assertion characterizing all the reachable states. Assertion $pend$ characterizes the states which can be reached from a reachable q-state by an r-free path. For each transition τ, assertion h_τ characterizes the states at which τ is *helpful*. These are the states s that have a ρ-successor s' such that τ is enabled on s and disabled on s', and the transition from s to s' leads to a progress towards the goal. This progress is observed by immediately reaching the goal or a decrease in the ranking function δ_τ, as stated in premise D5. The ranking functions δ_τ are used in order to measure progress towards the goal. The disabling of τ is often due to the taking of this transition, but may also be caused by some data condition turning false. We require decrease in ranking in both cases.

Premise D1 guarantees that any reachable q-state satisfies r or $pend$. Premise D2 guarantees that any successor of a $pend$-state also satisfies r or $pend$. Premise D3 guarantees that any $pend$-state has at least one transition which is helpful in this state. Premise D4 guarantees that ranking never increases on transitions between two $pend$-states. Note that, due to D2, every ρ-successor of a $pend$-state which has not reached

the goal is also a $pend$-state. Premise D5 guarantees that taking a step from an h_τ-state leads into a state which either already satisfies the goal r, or causes the rank δ_τ to decrease, or is again an h_τ-state. Premise D6 guarantees that all h_τ-states violate J_τ. Together, premises D5 and D6 imply that the computation cannot stay in h_τ forever without violating justice w.r.t J_τ. Therefore, the computation must eventually move to a $\neg h_\tau$-state, causing δ_τ to decrease. Since there are only finitely many δ_τ and until the goal is reached they monotonically decrease, we can conclude that eventually an r-state is reached.

3.2 Automatic Generation of the Auxiliary Constructs

We now proceed to show how all the auxiliary constructs necessary for the application of rule DISTRANK can be automatically generated. Note that we have to construct a symbolic version of these constructs, so that the rule can be applied to a generic N.

For simplicity, we assume that the response property we wish to establish only refers to the local variables of process $P[1]$. (The approach can be easily generalized to the case when the response property refers to the local variables of process $P[z]$, for an arbitrary $z \in [1..N]$.) A case in point is the verification of the property $at\_\ell_1[1] \Rightarrow \diamondsuit at\_\ell_2[1]$ for program TOKEN-RING. This property claims that every state in which process $P[1]$ is at location ℓ_1 is eventually followed by a state in which process $P[1]$ is at location ℓ_2. Assume that the parameter domain has the form $[0..m] \times N$.

Since the special process is $P[1]$, we would expect the constructs to have the symbolic forms $\varphi : \forall i.\varphi^{\mathrm{A}}(i)$ and $pend : pend^{\mathrm{A}}_{=1} \ \wedge \ \forall i \neq 1.pend^{\mathrm{A}}_{\neq 1}(i)$. For each $k \in [0..m]$, we need to compute $h^{\mathrm{A}}_k[1], \delta^{\mathrm{A}}_k[1]$, and the generic $h^{\mathrm{A}}_k[i], \delta^{\mathrm{A}}_k[i]$, which should be symbolic in i and apply for all i, $1 < i \leq N$. All generic constructs are allowed to refer to the global variables and to the variables local to $P[1]$. We will consider each of the auxiliary constructs and provide a methodology for its generation which will be illustrated on the case of program TOKEN-RING.

The construction uses the instantiation $S(N_0)$ for the cutoff value N_0 required in Claim 2.2. In our case, this yields $N_0 = 6$ as explained below. We denote by $\Theta_{_C}$ and $\rho_{_C}$ the initial condition and transition relation for $S(N_0)$. The construction begins by computing the *concrete* auxiliary constructs for $S(N_0)$ which we denote by $\varphi_{_C}, pend_{_C}$, $h^{\mathrm{C}}_k[j]$, and $\delta^{\mathrm{C}}_k[j]$. We then use *project-and-generalize* in order to derive the symbolic (*abstract*) versions of these constructs: $\varphi_{_A}, pend_{_A}, h^{\mathrm{A}}_k[j]$, and $\delta^{\mathrm{A}}_k[j]$.

Computing $\varphi_{_A}$: Compute the assertion $reach_{_C} = \Theta_{_C} \circ \rho^*_{_C}$ characterizing all states reachable by system $S(N_0)$. We take $\varphi^{\mathrm{A}}(i) = reach_{_C}[3 \mapsto i]$, which is obtained by projecting $reach_{_C}$ on index 3, and then generalizing 3 to i.

For example, in TOKEN-RING(6), $reach_{_C} = \bigwedge_{j=1}^{6}(at\_\ell_{0,1}[j] \vee tloc = j)$. The projection of $reach_{_C}$ on $j = 3$ yields $(at\_\ell_{0,1}[3] \vee tloc = 3)$. The generalization of 3 to i yields $\varphi^{\mathrm{A}}(i) : at\_\ell_{0,1}[i] \vee tloc = i$. Note that when we generalize, we should generalize not only the values of the variables local to $P[3]$ but also the case that the global variable, such as *tloc*, has the value 3. The choice of 3 as the generic value is arbitrary. Any other value would do as well, but we prefer indices different from $1, N$.

In this part we computed $\varphi^{\mathrm{A}}(i)$ as the generalization of 3 into i in $reach_{_C}$, which can be denoted as $\varphi^{\mathrm{A}}(i) = reach_{_C}[3 \mapsto i]$. In later parts we may need to generalize

two indices, such as $\alpha_{_A} = \alpha_{_C}[2 \mapsto i, 4 \mapsto j]$, where $\alpha_{_C}$ and $\alpha_{_A}$ are a concrete and abstract versions of some assertion α. The way we compute such abstractions over the state variables *tloc* and π of system TOKEN-RING is given by

$$\alpha_{_A}(tloc, \pi) \quad = i < j \ \wedge \ \exists tloc', \pi' : \alpha_{_C}(tloc', \pi') \ \wedge \ map(2, i, 4, j),$$

where,

$$map(2,i,4,j) = \left(\begin{array}{l} \pi[i] = \pi'[2] \ \wedge \ \pi[j] = \pi'[4] \qquad\qquad\qquad\qquad\qquad\qquad \wedge \\ tloc = i \iff tloc' = 2 \ \wedge \ tloc = j \iff tloc' = 4 \ \wedge \\ tloc < i \iff tloc' < 2 \ \wedge \ tloc < j \iff tloc' < 4 \end{array} \right)$$

Note that this computation is very similar to the symbolic computation of the predecessor of an assertion, where $map(2, i, 4, j)$ serves as a transition relation. Indeed, we use the same module used by a symbolic model checker for carrying out this computation.

Computing $pend_{_A}$: Compute the assertion $pend_{_C} = (reach_{_C} \wedge q \wedge \neg r) \circ (\rho_{_C} \wedge \neg r)^*$, characterizing all states which can be reached from a reachable $(q \wedge \neg r)$-state by an r-free path. Then we take $pend^{\mathrm{A}}_{=1} = pend_{_C}[1 \mapsto 1]$, and $pend^{\mathrm{A}}_{\neq 1}(i) = pend_{_C}[1 \mapsto 1, 3 \mapsto i]$.

Thus, for TOKEN-RING(6), $pend_{_C} = reach_{_C} \ \wedge \ at\_\ell_1[1]$. We therefore take $pend^{\mathrm{A}}_{=1}$: $at\_\ell_1[1]$ and $pend^{\mathrm{A}}_{\neq 1}(i) : at\_\ell_1[1] \ \wedge \ (at\_\ell_{0,1}[i] \vee tloc = i)$.

Computing $h^{\mathrm{A}}_k[i]$: First, we compute the concrete helpful assertions $h^{\mathrm{C}}_k[j]$. This is based on the following analysis. Assume that *set* is an assertion characterizing a set of states, and let J be some justice requirement. We wish to identify the subset of states ϕ within *set* for which the transition associated with J is an *escape* transition. That is, any application of this transition to a ϕ-state takes us out of *set*. Consider the fix-point equation:

$$\phi \quad = \quad set \ \wedge \ \neg J \ \wedge \ AX(\phi \ \vee \ \neg set) \tag{1}$$

The equation states that every ϕ-state must satisfy $set \wedge \neg J$, and that every successor of a ϕ-state is either a ϕ-state or lies outside of *set*. In particular, all J-satisfying successors of a ϕ-state do not belong to *set*. By taking the maximal solution of this fix-point equation, denoted $\nu\phi(set \wedge \neg J \ \wedge \ AX(\phi \vee \neg set))$, we compute the subset of states which are helpful for J within *set*.

Following is an algorithm which computes the concrete helpful assertions $\{h^{\mathrm{C}}_k[j]\}$ corresponding to the justice requirements $\{J^{\mathrm{C}}_k[j]\}$ of system $S(N_0)$. For simplicity, we will use $\tau \in \mathcal{T}(N_0)$ as a single parameter.

for each $\tau \in \mathcal{T}(N_0)$ **do** $h_\tau := 0$
$set := pend_{_C}$
for all $\tau \in \mathcal{T}(N_0)$ **satisfying** $\nu\phi(set \wedge \neg J_\tau \ \wedge \ AX(\phi \vee \neg set)) \neq 0$ **do**
$$\left[\begin{array}{l} h_\tau := h_\tau \ \vee \ \nu\phi(set \wedge \neg J_\tau \ \wedge \ AX(\phi \vee \neg set)) \\ set := set \ \wedge \ \neg h_\tau \end{array} \right]$$

The "**for all** $\tau \in \mathcal{T}(N_0)$" iteration terminates when it is no longer possible to find a $\tau \in \mathcal{T}(N_0)$ which satisfies the non-emptiness requirement. The iteration may choose the same τ more than once. When the iteration terminates, *set* is 0, i.e., for each of the states covered under $pend_{_C}$ there exists a helpful justice requirement which causes it to progress.

Having found the concrete $h_k^{\mathrm{C}}[j]$, we proceed to *project-and-generalize* as follows: for each $k \in [0..m]$, we let $h_k^{\mathrm{A}}[1] = h_k^{\mathrm{C}}[1][1 \mapsto 1]$ and $h_k^{\mathrm{A}}[i] = h_k^{\mathrm{C}}[3][1 \mapsto 1, 3 \mapsto i]$.

Applying this procedure to TOKEN-RING(6), we obtain the following symbolic helpful assertions:

	$h_k^{\mathrm{A}}[1]$	$h_k^{\mathrm{A}}[i], i > 1$
$k = 0$	0	$at\_\ell_1[1] \ \wedge \ at\_\ell_0[i] \ \wedge \ tloc = i$
$k = 1$	$at\_\ell_1[1] \ \wedge \ tloc = 1$	$at\_\ell_1[1] \ \wedge \ at\_\ell_1[i] \ \wedge \ tloc = i$
$k = 2$	0	$at\_\ell_1[1] \ \wedge \ at\_\ell_2[i] \ \wedge \ tloc = i$

Computing $\delta_k^{\mathrm{A}}[i]$: As before, we begin by computing the concrete ranking functions $\delta_k^{\mathrm{C}}[j]$. We observe that $\delta_k^{\mathrm{C}}[j]$ should equal 1 on every state for which $\tau_k[j]$ is helpful and should decrease from 1 to 0 on any transition that causes a helpful $\tau_k[j]$ to become unhelpful. Furthermore, $\delta_k^{\mathrm{C}}[j]$ can never increase. It follows that $\delta_k^{\mathrm{C}}[j]$ should equal 1 on every pending state from which there exists a pending path to a pending state satisfying $h_k^{\mathrm{C}}[j]$. Thus, we compute $\delta_k^{\mathrm{C}}[j] = pend_{_C} \ \wedge \ ((\neg r) \, E\mathcal{U} \, h_k^{\mathrm{C}}[j])$, where $E\mathcal{U}$ is the "existential-until" CTL operator. This formula identifies all states from which there exists an r-free path to a $(h_k^{\mathrm{C}}[j])$-state.

Having found the concrete $\delta_k^{\mathrm{C}}[j]$, we proceed to *project-and-generalize* as follows: for each $k \in [0..m]$, we let $\delta_k^{\mathrm{A}}[1] = \delta_k^{\mathrm{C}}[1][1 \mapsto 1]$ and $\delta_k^{\mathrm{A}}[i] = \delta_k^{\mathrm{C}}[3][1 \mapsto 1, 3 \mapsto i]$.

Applying this procedure to TOKEN-RING(6), we obtain the following abstract ranking functions:

$$
\begin{array}{l}
\delta_0^{\mathrm{A}}[1] = \delta_2^{\mathrm{A}}[1] : \quad 0 \\
\delta_1^{\mathrm{A}}[1] : at\_\ell_1[1] \\
\left.\begin{array}{l}
\delta_0^{\mathrm{A}}[i] : at\_\ell_1[1] \ \wedge \ (1 < tloc < i \ \wedge \ at\_\ell_{0,1}[i] \ \vee \ tloc = i) \\
\delta_1^{\mathrm{A}}[i] : at\_\ell_1[1] \ \wedge \ (1 < tloc < i \ \wedge \ at\_\ell_{0,1}[i] \ \vee \ tloc = i \ \wedge \ at\_\ell_1[i]) \\
\delta_2^{\mathrm{A}}[i] : at\_\ell_1[1] \ \wedge \ (1 < tloc < i \ \wedge \ at\_\ell_{0,1}[i] \ \vee \ tloc = i \ \wedge \ at\_\ell_{1,2}[i])
\end{array}\right\} \text{ for } i > 1
\end{array}
$$

3.3 Validating the Premises

Having computed internally the auxiliary constructs, and checking the invariantce of φ, it only remains to check that the six premises of rule DISTRANK are all valid for any value of N. Here we use the small model theorem stated in Claim 2.2 which allows us to check their validity for all values of $N \leq N_0$ for the cutoff value of N_0 which is specified in the theorem. First, we have to ascertain that all premises have the required AE form. For auxiliary constructs of the form we have stipulated in this Section, this is straightforward. Next, we consider the value of N_0 required in each of the premises, and take the maximum. Note that once φ is known to be inductive, we can freely add it to the left-hand-side of each premise, which we do for the case of Premises D5 and D6 that, unlike others, do not include any inductive component.

Usually, the most complicated premise is D2 and this is the one which determines the value of N_0. For program TOKEN-RING, this premise has the form (where we renamed the quantified variables to remove any naming conflicts):

$$((\forall a.pend(a)) \ \wedge \ (\exists i, i_1 \forall j, j_1.\psi(i, i_1, j, j_1))) \quad \rightarrow \quad r' \ \vee \ (\forall c.pend(c)),$$

which is logically equivalent to

$$\forall i, i_1, c \ \exists a, j, j_1.\Big(pend(a) \ \wedge \ \psi(i, i_1, j, j_1) \quad \rightarrow \quad r' \ \vee \ pend(c)\Big)$$

The **index** variables which are universally quantified or appear free in this formula are $\{i, i_1, c, tloc, 1, N\}$ whose count is 6. It is therefore sufficient to take $N_0 = 6$. Having determined the size of N_0, it is straightforward to compute the premises of $S(N)$ for all $N \leq N_0$ and check that they are valid, using BDD symbolic methods.

The same form of auxiliary constructs can be used in order to automatically verify algorithm BAKERY(N), for every N. However, this requires the introduction of an auxiliary variable *min_id* into the system, which is the index of the process which holds the ticket with minimal value. In a future work we will present an extension of the method which enables us to verify BAKERY with no additional auxiliary variables.

4 Cases Requiring an Existential Invariant

In some cases, assertions of the form $\forall i.u(i)$ are insufficient for capturing all the relevant features of the constructs φ^{A} and $pend^{\mathrm{A}}$, and we need to consider assertions of the form $\forall i.u(i) \ \wedge \ \exists j.e(j)$. Consider, for example, program CHANNEL-RING, presented in Fig. 3.

in N : **natural where** $N > 1$
chan : **array**$[1..N]$ **of boolean where** $chan[i] = (i = 2)$

$$\Big\|_{i=1}^{N} P[i] :: \Big[\begin{array}{l}\textbf{loop forever do}\\ \quad \Big[\begin{array}{l}\ell_0 : \textbf{if } chan[i] \textbf{ then } (chan[i], chan[i \oplus_N 1]) := (0, 1)\\ \quad\quad \textbf{go to } \{\ell_0, \ell_1\}\\ \ell_1 : \textbf{await } chan[i]\\ \ell_2 : \textbf{Critical}\end{array}\Big]\end{array}\Big]$$

Fig. 3. Program CHANNEL-RING

In this program the location of the token is identified by the index i such that $chan[i] = 1$. Computing the universal invariant according to the previous methods we obtain φ^{A} : $\forall i.(at\_\ell_{0,1} \ \vee \ chan[i])$, which is inductive but insufficient in order to establish the existence of a helpful transition for every pending state.

Using a recent extension to the invariant-generation method, we can now derive invariants of the form $\forall i.u(i) \ \wedge \ \exists j.e(j)$. Applying this method to the above example, we obtain

$$\varphi^{\mathrm{A}} : \forall i.(at\_\ell_{0,1} \ \vee \ chan[i]) \ \wedge \ \exists j.chan[j]$$

Using this extended form of an invariant for both φ^{A} and $pend^{\mathrm{A}}$, we can complete the proof of program CHANNEL-RING using the methods of Section 3.

We provide a sketch of the extension which enables the computation of an invariant such as $\forall i.u(i) \ \wedge \ \exists j.e(j)$. As before, we pick a value N_0, instantiate $S(N_0)$ and use the "invisible invariant" method to derive an inductive assertion $\forall i.u(i)$. As part of this computation we also compute $reach_C$ which captures all the states reachable in $S(N_0)$. Since we believe that there is still an existential invariant as part of $reach_C$, the assertion $\forall i.u(i)$ must be a strict over-approximation of $reach_C$. Below, we list the sequence of steps we take in order to isolate the assertion $e(j)$. On the right of this sequence of steps, we list the results of these computations for the case that $reach_C$ equals precisely the conjunction $\bigwedge_{i=1}^{N_0} u(i) \ \wedge \ \bigvee_{j=1}^{N_0} e(j)$.

Algorithm	Results when $reach_C = \bigwedge_i u(i) \ \wedge \ \bigvee_j e(j)$
$\alpha_1 := \bigwedge_i u(i) \ \wedge \ \neg reach_C$	$\alpha_1 = \bigwedge_i u(i) \ \wedge \ \bigwedge_j \neg e(j)$
$\alpha_2 := \alpha_1[1 \mapsto k]$	$\alpha_2 = u(k) \ \wedge \ \neg e(k)$
$\alpha_3 := \neg\alpha_2$	$\alpha_3 = u(k) \rightarrow e(k)$

Thus, we compute $\exists k.\alpha_3(k)$ as the candidate for an existential invariant. Note that, while we did not succeeded in precisely isolating $e(k)$, we computed instead the implication $e(k) \rightarrow u(k)$ and, since $\forall i.u(i)$ is invariant, and the **index** range in nonempty, $\exists k.e(k)$ is invariant iff $\exists k.e(k) \rightarrow u(k)$ is.

This technique of obtaining an existential conjunct to an auxiliary assertion can be used for other auxiliary constructs. Applying the method of invisible ranking, with the new addition, to program CHANNEL-RING and the response property $at\_\ell_1[1] \Rightarrow \Diamond at\_\ell_2[1]$, we obtain, for example, $pend^{\mathrm{A}} : at\_\ell_1[1] \ \wedge \ \varphi^{\mathrm{A}}$, and for $i > 1$, $h_2^{\mathrm{A}}[i] : \exists j \leq i.chan[j] \ \wedge \ at\_\ell_1[i] \ \wedge \ at\_\ell_1[1]$. Thus, Premise D3 becomes:

$$at\_\ell_1[1] \ \wedge \ \forall i.(at\_\ell_{0,1} \ \vee \ chan[i]) \ \wedge \ \exists j.chan[j] \ \rightarrow \ at\_\ell_1[1] \ \wedge \ \exists j.chan[j]$$

which is obviously valid.

5 The Bakery Algorithm

As the final example of the application of the invisible-ranking method we consider program BAKERY, presented in Fig. 4. This version is a variant of Lamport's original Bakery Algorithm that offers a solution of the mutual exclusion problem for any N processes. The operation "$y :=$maximal value to $y[i]$ while preserving order of elements" is defined by

$$\forall j,k : y'[j] < y'[i] \ \wedge \ (y[j] = 0 \leftrightarrow y'[j] = 0) \ \wedge \ (y[j] < y[k] \leftrightarrow y'[j] < y'[k])$$

where i, j, and k are mutually distinct. This assignment is in general non-deterministic. Here, π, the program location array, is of type **index** $\mapsto$ **bool**, and y, the "ticket" array, is of type **index** $\mapsto$ **par_data**.

The program contains the auxiliary variable *min_id* which is expected to hold the index of a process whose y value is minimal among all the positive y-values. The **maintaining** construct implies that this variable is updated, if necessary, whenever some

$$\begin{array}{l} \quad\quad\quad\quad\quad\quad \textbf{in} \quad N \quad : \textbf{natural where } N > 1 \\ \quad\quad\quad\quad\quad\quad \textbf{local} \quad y \quad : \textbf{array } [1..N] \textbf{ of } [0..N] \textbf{ where } y = 0 \\ \quad\quad\quad\quad\quad\quad\quad\quad \mathit{min\_id} : \textbf{natural where } \mathit{min\_id} = 1 \\ \mathop{\Big|\Big|}_{i=1}^{N} P[i] :: \left[\begin{array}{l} \textbf{loop forever do} \\ \quad \left[\begin{array}{l} 0 : \textbf{NonCritical} \\ 1 : y := \text{maximal value to } y[i] \text{ while preserving order of elements} \\ 2 : \textbf{await } \forall j : (y[j] = 0 \ \vee \ y[j] < y[i]) \\ 3 : \textbf{Critical} \\ 4 : y[i] := 0 \end{array} \right] \\ \textbf{maintaining } \forall j : y[j] = 0 \ \vee \ 0 < y[\mathit{min\_id}] \leq y[j] \end{array} \right] \end{array}$$

Fig. 4. Program BAKERY

y variables change their values. Already in [20] we pointed out that in some cases, it is necessary to add auxiliary variables in order to find inductive assertions with fewer indices. This version of BAKERY illustrates the case that such auxiliary variables may also be needed in the case of the invisible ranking method.

The property we wish to verify for this parameterized system is $\mathit{at\_}\ell_1[z] \implies \Diamond \mathit{at\_}\ell_3[z]$ which implies accessibility for an arbitrary process $P[z]$.

Having the auxiliary variable *min_id* as part of the system variables, we can proceed with the computation of the auxiliary constructs following the recipes explained in Section 3: After some simplifications, we can present the automatically derived constructs as follows:

$$\begin{array}{ll} \varphi_{_A} : & \forall i : (\mathit{at\_}\ell_{0,1}[i] \leftrightarrow y[i] = 0) \ \wedge \ (\mathit{at\_}\ell_{3,4}[i] \rightarrow \mathit{min\_id} = i) \quad \wedge \\ & \forall i \neq j : \left(\begin{array}{l} (\mathit{min\_id} = i \ \rightarrow \ y[j] > y[i] \wedge y[i] \neq 0 \ \vee \ y[j] = 0) \ \wedge \\ (y[i] = y[j] \ \rightarrow \ y[i] = 0) \end{array} \right) \\ \mathit{pend}_{_A} : & \varphi_{_A} \ \wedge \ \mathit{at\_}\ell_{1,2}[z] \\ (h_\ell[j])_{_A} : & \left(\begin{array}{c|l|l} \ell & \text{For } j = z & \text{For } j \neq z \\ \hline 1 & \mathit{at\_}\ell_1[z] & 0 \\ 2 & \mathit{at\_}\ell_2[z] \ \wedge \ \mathit{min\_id} = z & \mathit{at\_}\ell_2[z] \ \wedge \ \mathit{min\_id} = j \ \wedge \ \mathit{at\_}\ell_2[j] \\ 3 & 0 & \mathit{at\_}\ell_2[z] \ \wedge \ \mathit{at\_}\ell_3[j] \\ 4 & 0 & \mathit{at\_}\ell_2[z] \ \wedge \ \mathit{at\_}\ell_4[j] \end{array} \right) \\ (\delta_\ell[j])_{_A} : & \left(\begin{array}{c|l|l} \ell & \text{For } j = z & \text{For } j \neq z \\ \hline 1 & \mathit{at\_}\ell_1[z] & 0 \\ 2 & \mathit{at\_}\ell_{1,2}[z] & \mathit{at\_}\ell_1[z] \ \vee \ \mathit{at\_}\ell_2[z] \ \wedge \ \mathit{at\_}\ell_2[j] \ \wedge \ y[z] > y[j] \\ 3 & 0 & \mathit{at\_}\ell_1[z] \ \vee \ \mathit{at\_}\ell_2[z] \ \wedge \ \mathit{at\_}\ell_{2,3}[j] \ \wedge \ y[z] > y[j] \\ 4 & 0 & \mathit{at\_}\ell_1[z] \ \vee \ \mathit{at\_}\ell_2[z] \ \wedge \ \mathit{at\_}\ell_{2..4}[j] \ \wedge \ y[z] > y[j] \end{array} \right) \end{array}$$

Using these derived auxiliary constructs, we can verify the validity of the premises of rule DISTRANK over $S(5)$ and conclude that the property of accessibility holds for every value of N.

6 Conclusion and Future Work

The paper showed how to extend the method of invisible invariants to a fully automatic proof of parameterized systems $S(N)$ for any value of N. We presented rule DISTRANK, a distributed ranking proof rule that allows an automatic computation of helpful assertions and ranking functions, which can be also automatically validated.

The current method only deals with helpful assertions which depend on a single parameter. In a future work, we plan to present a rule which enables us to deal with helpful assertions of the form $h_i : \forall j \neq i.\psi(i,j)$, and extend its applicability to systems with strong fairness requirements.

References

1. K. R. Apt and D. Kozen. Limits for automatic program verification of finite-state concurrent systems. *Information Processing Letters*, 22(6), 1986.
2. T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. Zuck. Parameterized verification with automatically computed inductive assertions. In CAV'01, pages 221–234. LNCS 2102, 2001.
3. K. Baukus, Y. Lakhnesche, and K. Stahl. Verification of parameterized protocols. *Journal of Universal Computer Science*, 7(2):141–158, 2001.
4. N. Bjørner, I. Browne, E. Chang, M. Colón, A. Kapur, Z. Manna, H. Sipma, and T. Uribe. STeP: The Stanford Temporal Prover, User's Manual. Technical Report STAN-CS-TR-95-1562, Computer Science Department, Stanford University, 1995.
5. N. Bjørner, I. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. In 1^{st} *Intl. Conf. on Principles and Practice of Constraint Programming*, pages 589–623. LNCS 976, 1995.
6. E. Clarke, O. Grumberg, and S. Jha. Verifying parametrized networks using abstraction and regular languages. In *(*CONCUR'95), pages 395–407. LNCS 962, 1995.
7. M. Colon and H. Sipma. Practical methods for proving program termination. In CAV'02, pages 442–454. LNCS 2404, 2002.
8. E. Emerson and V. Kahlon. Reducing model checking of the many to the few. In *(*CADE'00), pages 236–255, 2000.
9. E. A. Emerson and K. S. Namjoshi. Reasoning about rings. In POPL'95, 1995.
10. E. Gribomont and G. Zenner. Automated verification of szymanski's algorithm. In TACAS'98, pages 424–438. LNCS 1384, 1998.
11. V. Gyuris and A. P. Sistla. On-the-fly model checking under fairness that exploits symmetry. In CAV'97. LNCS 1254, 1997.
12. B. Jonsson and M. Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In TACAS'00. LNCS 1785, 2000.
13. Y. Kesten and A. Pnueli. Control and data abstractions: The cornerstones of practical formal verification. *Software Tools for Technology Transfer*, 4(2):328–342, 2000.
14. Y. Kesten, A. Pnueli, E. Shahar, and L. Zuck. Network invariants in action. In CONCUR'02, pages 101–105. LNCS 2421, 2002.
15. C. S. Lee, N. Jones, and A. Ben-Amram. The size-change principle for program termination. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 81–92. ACM Press, 2001.
16. D. Lesens, N. Halbwachs, and P. Raymond. Automatic verification of parameterized linear networks of processes. In POPL'97, 1997.
17. Z. Manna and A. Pnueli. Completing the temporal picture. *Theoretical Comput. Sci.*, 83(1):97–130, 1991.

18. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.
19. K. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In CAV'98, pages 110–121. LNCS 1427, 1998.
20. A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In TACAS'01, pages 82–97. LNCS 2031, 2001.
21. A. Pnueli, J. Xu, and L. Zuck. Liveness with $(0, 1, \infty)$-counter abstraction. In CAV'02, pages 107–122. LNCS 2404, 2002.
22. N. Shankar, S. Owre, and J. Rushby. The PVS proof checker: A reference manual. Technical report, Comp. Sci.,Laboratory, SRI International, Menlo Park, CA, 1993.
23. L. Zuck, A. Pnueli, and Y. Kesten. Automatic verification of free choice. In *Proc. of the 3^{r}d workshop on Verification, Model Checking, and Abstract Interpretation*, pages 208–224. LNCS 2294, 2002.

A Complete Method for the Synthesis of Linear Ranking Functions

Andreas Podelski and Andrey Rybalchenko

Max-Planck-Institut für Informatik
Saarbrücken, Germany

Abstract. We present an automated method for proving the termination of an unnested program loop by synthesizing linear ranking functions. The method is complete. Namely, if a linear ranking function exists then it will be discovered by our method. The method relies on the fact that we can obtain the linear ranking functions of the program loop as the solutions of a system of linear inequalities that we derive from the program loop. The method is used as a subroutine in a method for proving termination and other liveness properties of more general programs via transition invariants; see [PR03].

1 Introduction

The verification of termination and other liveness properties of programs is a difficult problem. It requires the discovery of invariants and ranking functions to prove the termination of program loops.

We present a complete and efficient method for the synthesis of linear ranking functions for unnested program loops whose guards and update statements use linear arithmetic expressions. We have implemented the method. Preliminary experiments show that the method is efficient not only in theory but also in practice.

Roughly, the method works as follows. Given a program loop for which we want to find a linear ranking function, we construct a corresponding system of linear inequalities over rationals. As we show, the solutions of this system encode the linear ranking functions of the program loop. That is, we can check the existence of a linear ranking function by constraint solving. If it exists, a linear ranking function can be constructed from a solution of the system of linear inequalities, a solution that we obtain by constraint solving. If the system has no solutions then (and only then) a linear ranking function does not exist. As a consequence of our approach, one can use existing highly-optimized tools for linear programming as the engine in a complete method (to our knowledge the first) for the synthesis of linear ranking functions.

We admit unnested program loops with nondeterministic update statements. This is potentially useful to model `read` statements. It is strictly required in the context where we employ our method, described next.

In a work described elsewhere [PR03], we show that one can reduce the test of termination and other liveness properties (in the presence of fairness

B. Steffen and G. Levi (Eds.): VMCAI 2004, LNCS 2937, pp. 239–251, 2004.

assumptions) to the test of termination of unnested program loops. That is, we use the algorithm described in this paper as a subroutine in the software model checking method for liveness properties via transition invariants proposed in [PR03]. The experiments that we present in this paper stem from this context.

2 Unnested Program Loops

We formalize the notion of unnested program loops by a class of programs that are built using a single "while" statement and that satisfy the following conditions:

- the loop condition is a conjunction of atomic propositions,
- the loop body may only contain update statements,
- all update statements are executed simultaneously.

We call this class *simple while programs*. Pseudo-code notation for the programs of this class is given below.

while $(Cond_1$ **and** $\ldots$ **and** $Cond_m)$ **do**
    Simultaneous Updates
od

We consider the subclass of simple while programs built using linear arithmetic expressions over program variables.

Definition 1. *A linear arithmetic simple while (LASW) program over the tuple of program variables $x = (x_1, \ldots, x_n)$ is a simple while program such that:*

- *program variables have integer domain,*
- *every atomic proposition in the loop condition is a linear inequality over (unprimed) program variables:*

$$c_1 x_1 + \cdots + c_n x_n \leq c_0,$$

- *every update statement is a linear inequality over unprimed and primed program variables*

$$a'_1 x'_1 + \cdots + a'_n x'_n \leq a_1 x_1 + \cdots + a_n x_n + a_0.$$

Note that we allow the left-hand side of an update statement to be a linear expression over program variables, and that an update can be nondeterministic, e.g., $x' + y' \leq x + 2y - 1$. This is necessary, because we use simple while programs, and LASW programs in particular, to approximate the transitive closure of a transition relation (see Section 4).

We define a *program state* to be a valuation of program variables. The set of all program states is called the *program domain*. The *transition relation* denoted by the loop body of an LASW program is the set of all pairs of program states (s, s') such that the state s satisfies the loop condition, and (s, s') satisfies

each update statement. A *trace* is a sequence of states such that each pair of consecutive states belongs to the transition relation of the loop body.

We observe that the transition relation of a LASW program can be expressed by a system of inequalities over unprimed and primed program variables. The translation procedure is straightforward. For the rest of this paper, we assume that an LASW program over the tuple of program variables $x = (x_1, \ldots, x_n)$ (treated as a column vector) can be represented by the system

$$(AA')\begin{pmatrix} x \\ x' \end{pmatrix} \leq b$$

of inequalities. We identify an LASW program with the corresponding system of inequalities.

Example 1. The following program loop with nondeterministic updates

while $(i - j \geq 1)$ **do**
$\quad (i, j) := (i - Nat, j + Pos)$
od

is represented by the following system of inequalities.

$$\begin{aligned} -i + j &\leq -1 \\ -i + i' &\leq 0 \\ j - j' &\leq -1 \end{aligned}$$

Note that the relations between program variables denoted by the nondeterministic update statements $i := i - Nat$ and $j := j + Pos$, where Nat and Pos stand for any nonnegative and positive integer number respectively, can be expressed by the inequalities $i' \leq i$ and $j' \geq j + 1$.

3 The Algorithm

We say that a simple while program is *terminating* if the program domain is well-founded by the transition relation of the loop body of the program, i.e., if there is no infinite sequence $\{s_i\}_{i=1}^{\infty}$ of program states such that each pair (s_i, s_{i+1}), where $i \geq 1$, is an element of the transition relation.

The following theorem allows us to use linear programming over rationals to test the existence of a linear ranking function, and thus to test a sufficient condition for termination of LASW programs. The corresponding algorithm is shown in Figure 1.

Theorem 1. *A linear arithmetic simple while program given by the system* $(AA')\binom{x}{x'} \leq b$ *is terminating if there exist nonnegative vectors over rationals*

input
program $(AA')\binom{x}{x'} \leq b$
begin
if exists rational-valued λ_1 **and** λ_2 **such that**
$\lambda_1, \lambda_2 \geq 0$
$\lambda_1 A' = 0$
$(\lambda_1 - \lambda_2)A = 0$
$\lambda_2(A + A') = 0$
$\lambda_2 b < 0$
then
return("Program Terminates")
else
return("Linear ranking function does not exist")
end.

Given λ_1 and λ_2, solutions of the systems above, define $r \stackrel{\text{def}}{=} \lambda_2 A'$, $\delta_0 \stackrel{\text{def}}{=} -\lambda_1 b$, and $\delta \stackrel{\text{def}}{=} -\lambda_2 b$. A linear ranking function ρ is defined by

$$\rho(x) \stackrel{\text{def}}{=} \begin{cases} rx & \text{if exists } x' \text{ such that } (AA')\binom{x}{x'} \leq b, \\ \delta_0 - \delta & \text{otherwise.} \end{cases}$$

Fig. 1. Termination Test and Synthesis of Linear Ranking Functions.

λ_1 and λ_2 such that the following system is satisfiable.

$$\lambda_1 A' = 0 \tag{1a}$$
$$(\lambda_1 - \lambda_2)A = 0 \tag{1b}$$
$$\lambda_2(A + A') = 0 \tag{1c}$$
$$\lambda_2 b < 0 \tag{1d}$$

Proof. Let the pair of nonnegative (row) vectors λ_1 and λ_2 be a solution of the system (1a)–(1d). For every x and x' such that $(AA')\binom{x}{x'} \leq b$, by assumption that $\lambda_1 \geq 0$, we have $\lambda_1(AA')\binom{x}{x'} \leq \lambda_1 b$. We carry out the following sequence of transformations.

$$\begin{aligned} \lambda_1(Ax + A'x') &\leq \lambda_1 b \\ \lambda_1 Ax + \lambda_1 A'x' &\leq \lambda_1 b \\ \lambda_1 Ax &\leq \lambda_1 b && \text{by (1a)} \\ \lambda_2 Ax &\leq \lambda_1 b && \text{by (1b)} \\ -\lambda_2 A'x &\leq \lambda_1 b && \text{by (1c)} \end{aligned}$$

From the assumption $\lambda_2 \geq 0$ follows $\lambda_2(AA')\binom{x}{x'} \leq \lambda_2 b$. Then, we continue with

$$\begin{aligned} \lambda_2(Ax + A'x') &\leq \lambda_2 b \\ \lambda_2 Ax + \lambda_2 A'x' &\leq \lambda_2 b \\ -\lambda_2 A'x + \lambda_2 A'x' &\leq \lambda_2 b \qquad \text{by (1c)} \end{aligned}$$

We define $r \stackrel{\text{def}}{=} \lambda_2 A'$, $\delta_0 \stackrel{\text{def}}{=} -\lambda_1 b$, and $\delta \stackrel{\text{def}}{=} -\lambda_2 b$. Then, we have $rx \geq \delta_0$ and $rx' \leq rx - \delta$ for all x and x' such that $(AA')\binom{x}{x'} \leq b$. Due to (1d) we have $\delta > 0$.

We define a function ρ as shown in Figure 1. Any program trace induces a strictly descending sequence of values under ρ that is bounded from below, and the difference between two consecutive values is at least δ. Since no such infinite sequence exists, the program is terminating. □

The theorem above states a sufficient condition for termination. We observe that if the condition applies then a linear ranking function, i.e., a linear arithmetic expression over program variables which maps program states into a well-founded domain, exists. The following theorem states that our termination test is complete for the programs with linear ranking functions.

Theorem 2. *If there exists a linear ranking function for the linear arithmetic simple while program with nonempty transition relation then the termination condition of Theorem 1 applies.*

Proof. Let the vector r together with the constants δ_0 and $\delta > 0$ define a linear ranking function. Then, for all pairs x and x' such that $(AA')\binom{x}{x'} \leq b$ we have $rx \geq \delta_0$ and $rx' \leq rx - \delta$.

By the non-emptiness of the transition relation, the system $(AA')\binom{x}{x'} \leq b$ has at least one solution. Hence, we can apply the 'affine' form of Farkas' lemma (in [Sch86]), from which follows that there exists δ_0' and δ' such that $\delta_0' \geq \delta_0$, $\delta' \geq \delta$, and each of the inequalities $-rx \leq -\delta_0'$ and $-rx + rx' \leq -\delta'$ is a nonnegative linear combination of the inequalities of the system $(AA')\binom{x}{x'} \leq b$. This means that there exist nonnegative rational-valued vectors λ_1 and λ_2 such that

$$\begin{aligned} \lambda_1(AA')\tbinom{x}{x'} &= -rx \\ \lambda_1 b &= -\delta_0' \end{aligned}$$

and

$$\begin{aligned} \lambda_2(AA')\tbinom{x}{x'} &= -rx + rx' \\ \lambda_2 b &= -\delta'. \end{aligned}$$

After multiplication and simplification we obtain

$$\begin{aligned} \lambda_1 A &= -r & \lambda_1 A' &= 0 \\ \lambda_2 A &= -r & \lambda_2 A' &= r, \end{aligned}$$

from which equations (1a)–(1c) follow directly. Since $\delta' \geq \delta > 0$, we have $\lambda_2 b < 0$, i.e., the equation (1d) holds. □

The following corollary is an immediate consequence of Theorems 1 and 2.

Corollary 1. *Existence of linear ranking functions for linear arithmetic simple while programs with nonempty transition relation is decidable in polynomial time.*

Not every LASW program has a linear ranking function (see the following example).

Example 2. Consider the following program.

$$\begin{array}{l}\textbf{while } (x \geq 0) \textbf{ do}\\ \quad x := -2x + 10\\ \textbf{od}\end{array}$$

The program is terminating, but it does not have a linear ranking function. For termination proof consider the following ranking function into the domain $\{0, \ldots, 3\}$ well-founded by the less-than relation $<$.

$$\rho(x) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } x \in \{0,1,2\},\\ 2 & \text{if } x \in \{4,5\},\\ 3 & \text{if } x = 3,\\ 0 & \text{otherwise.}\end{cases}$$

It can be easily tested that the system (1a)–(1d) is not satisfiable for the LASW program

$$\begin{pmatrix} -1 & 0\\ 2 & 1\\ -2 & -1\end{pmatrix}\begin{pmatrix} x\\ x'\end{pmatrix} \leq \begin{pmatrix} 0\\ 10\\ -10\end{pmatrix}.$$

By Theorem 2, this implies that no linear ranking function exists for the program above. □

The following example illustrates an application of the algorithm based on Theorem 1.

Example 3. We prove termination of the LASW program from Example 1. The program translates to the system $(AA')\binom{x}{x'} \leq b$, where:

$$A \stackrel{\text{def}}{=} \begin{pmatrix} -1 & 1\\ -1 & 0\\ 0 & 1\end{pmatrix}, \qquad A' \stackrel{\text{def}}{=} \begin{pmatrix} 0 & 0\\ 1 & 0\\ 0 & -1\end{pmatrix},$$

$$x \stackrel{\text{def}}{=} \begin{pmatrix} i\\ j\end{pmatrix}, \qquad b \stackrel{\text{def}}{=} \begin{pmatrix} -1\\ 0\\ -1\end{pmatrix}.$$

Let $\lambda_1 = (\lambda'_1, \lambda'_2, \lambda'_3)$ and $\lambda_2 = (\lambda''_1, \lambda''_2, \lambda''_3)$. The system (1a)–(1d) is feasible, it has the following solutions:

$$\begin{aligned} \lambda'_2 = \lambda'_3 = \lambda''_1 = 0, \\ \lambda'_1 = \lambda''_2 = \lambda''_3, \\ \lambda'_1, \lambda''_2, \lambda''_3 > 0. \end{aligned}$$

Since the system is feasible the program is terminating. We construct a linear ranking function following the algorithm in Figure 1. We define $r \stackrel{\text{def}}{=} \lambda_2 A'$, $\delta_0 \stackrel{\text{def}}{=} -\lambda_1 b$, and $\delta \stackrel{\text{def}}{=} -\lambda_2 b$, and obtain $r = (\lambda'_1 \ \ -\lambda'_1)$, $\delta_0 = \delta = \lambda'_1$. Taking $\lambda'_1 = 1$ we obtain the following ranking function.

$$\rho(i,j) = \begin{cases} i - j & \text{if } i - j \geq 1, \\ 0 & \text{otherwise.} \end{cases}$$

4 Application to General Programs

In this section we illustrate how our method for proving termination of program loops can be used in the software model checking method for liveness properties via transition invariants proposed in [PR03]. That method applies to general-purpose programs (imperative, concurrent, ...); it is different from other approaches to special classes of infinite-state systems, e.g. [BS99]. We then provide experimental results obtained by applying the transition invariants approach for proving termination of singular value decomposition program.

Software model checking for liveness properties is a new approach for the automated verification of liveness properties of infinite-state systems by the computation of *transition invariants*. A transition invariant is an over-approximation of the transitive closure of the transition relation of the system. The presentation of a transition invariant as nothing but a finite set of unnested program loops. One can characterize the validity of a liveness property via the existence of transition invariants [PR03]. Namely, the liveness property is valid if each of the unnested program loops is terminating.

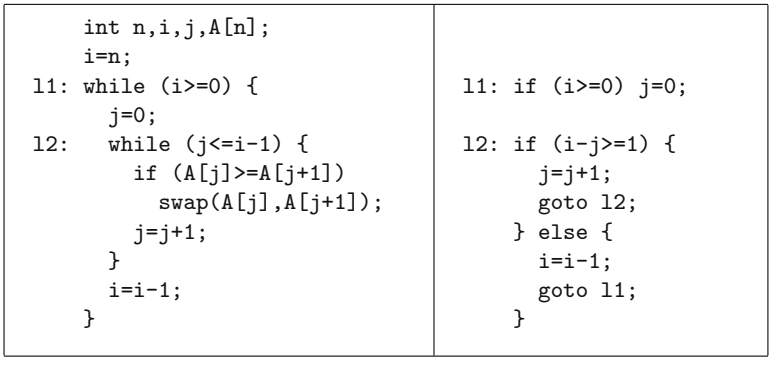
That is, the general method for the verification of liveness properties described in [PR03] is parameterized by an algorithm that tests whether each unnested program loop in the transition invariant is terminating. i.e., a procedure implementing a termination test for simple while programs.

Proving termination of simple while programs built using linear arithmetic expressions is required for the verification of a large class of software systems, e.g., liveness properties for mutual exclusion protocols (bakery, ticket), termination proofs of imperative programs (sorting algorithms, numerical algorithms dealing with matrices).

4.1 Sorting Program

This example illustrates the approach from [PR03] and the role of simple while programs.

We consider the program shown in Figure 2 implementing a sorting algorithm. For legibility, we concentrate on the skeleton shown on the right, which



```
    int n,i,j,A[n];
    i=n;
l1: while (i>=0) {
      j=0;
l2:   while (j<=i-1) {
        if (A[j]>=A[j+1])
          swap(A[j],A[j+1]);
        j=j+1;
      }
      i=i-1;
    }
```

```
l1: if (i>=0) j=0;

l2: if (i-j>=1) {
      j=j+1;
      goto l2;
    } else {
      i=i-1;
      goto l1;
    }
```

Fig. 2. Sorting program and its skeleton.

consists of the statements `st1`, `st2`, `st3`.

```
l1: if (i>=0)   { (i,j):=(i,0);   goto l2; }    /* st1 */
l2: if (i-j>=1) { (i,j):=(i,j+1); goto l2; }    /* st2 */
l2: if (i-j<1)  { (i,j):=(i-1,j); goto l1; }    /* st3 */
```

We read, for example, the first program statement as: if the current program location is labeled by `l1` and the "`if`" condition is satisfied then update the variables according to the update expressions and change the current label label to `l2`. Note that the updates are performed simultaneously ("concurrent" assignments in [Dij76]).

Each of the 'simple' programs below must be read as a one-line program.

```
l1: if (true)   { (i,j):=(Any,Any);      goto l2; }   /* a1 */
l2: if (true)   { (i,j):=(Any,Any);      goto l1; }   /* a2 */
l1: if (i>=0)   { (i,j):=(i-Pos,Any);    goto l1; }   /* a3 */
l2: if (i>=0)   { (i,j):=(i-Pos,Any);    goto l2; }   /* a4 */
l2: if (i-j>=1) { (i,j):=(i-Nat,j+Pos); goto l2; }   /* a5 */
```

Note the nondeterministic update expressions, e.g., after execution of `i:=Any` the value of variable `i` could by any integer, the update `i:=i-Pos` decrements the value of `i` by at least one.

We notice that `st1` is approximated by `a1`, `st2` by `a5` and `st3` by `a2`. This means that every transition induced by execution of the statement `st1` can also be achieved executing a single step of `a1`. In fact, every sequence of program statements is approximated by one of `a1`, ..., `a5`. We say that the set {`a1`, ..., `a5`} is a transition invariant in our terminology.

For example, every sequence of program statements that leads from `l2` to `l2` is approximated by `a4` if it passes through `l1`, and by `a5` otherwise. The following table assigns to each 'simple' program the set of sequences of program statements that it approximates. All non-assigned sequences are not feasible.

a1	st1(st2|st3st1)*
a2	(st2|st3st1)*st3
a3	st1(st2|st3st1)*st3
a4	(st2|st3st1)*st3st1(st2|st3st1)*
a5	st2+

According to the formal development in [PR03], the transition invariant above is 'strong enough' to prove termination, which means: each of its 'simple' programs, viewed in isolation, is terminating.

Termination is obvious for 'simple' programs that do not refer to a loop in the control flow graph, like the 'simple' programs `a1` and `a2`. The 'simple' programs of the form

```
ln: if (cond) { updates;  goto ln; }
```

translate to a while loop

```
ln: while (cond) { updates; }.
```

The 'simple' programs that translate to a while loop are in fact simple while programs whose termination proofs we study in this paper.

Next, we describe an application of the transition invariants method with termination test in Figure 1 as a subroutine for checking whether a transition invariant is strong enough. We prove termination of a program implementing singular value decomposition algorithm.

4.2 Program with Unbounded Nondeterminism

We consider the program shown in Figure 3. It has a nondeterministic choice at the location labeled by `l`. The value of the variable y is chosen nondeterministically in the first branch. Termination proof for this program requires a lexicographic ranking function. The program translates to the statements `st1` and `st2`:

```
l: if (x>=0)   { (x,y):=(x-1,Any);   goto l; }   /* st1 */
l: if (y>=0)   { (x,y):=(x,y-1);     goto l; }   /* st2 */
```

The transition invariant computed by our tool consists of the following 'simple' programs.

```
l: if (x>=0)   { (x,y):=(x-Pos,Any);   goto l; }   /* a1 */
l: if (y>=0)   { (x,y):=(Any,y-Pos);   goto l; }   /* a2 */
```

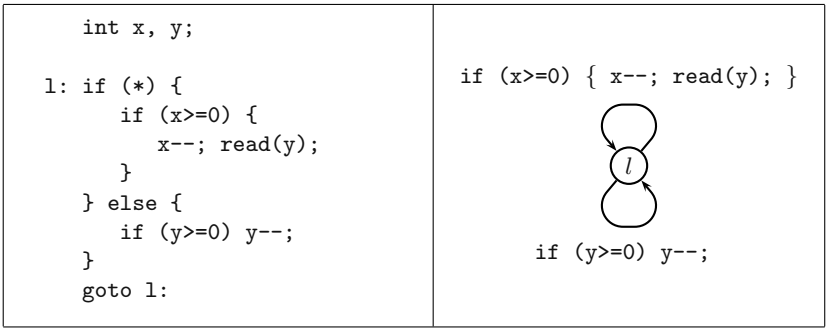


Fig. 3. Program with unbounded nondeterminism.

Both 'simple' programs, viewed in isolation, are terminating. Hence, the program with unbounded nondeterminism, shown in Figure 3 is terminating.

4.3 Singular Value Decomposition Program

We considered an algorithm for constructing the singular value decomposition (SVD) of a matrix. SVD is a set of techniques for dealing with sets of equations or matrices that are either singular or numerically very close to singular [PTVF92]. A matrix A is singular if it does not have a matrix inverse A^{-1} such that $AA^{-1} = I$, where I is the identity matrix.

Singular value decomposition of the matrix A whose number of rows m is greater or equal to its number of columns n is of the form

$$A = UWV^T,$$

where U is an $m \times n$ column-orthogonal matrix, W is an $n \times n$ diagonal matrix with positive or zero elements (called singular values), and the transpose matrix of an $n \times n$ orthogonal matrix V. Orthogonality of the matrices U and V means that their columns are orthogonal, i.e.,

$$U^T U = VV^T = I.$$

The SVD decomposition always exists, and is unique up to permutation of the columns of U, elements of W and columns of V, or taking linear combinations of any columns of U and V whose corresponding elements of W are exactly equal.

SVD can be used in numerically difficult cases for solving sets of equations, constructing an orthogonal basis of a vector space, or for matrix approximation [PTVF92].

We proved termination of a program implementing the SVD algorithm based on a routine described in [GVL96]. The program was taken from [PTVF92]. It is written in C and contains 163 lines of code with 42 loops in the control-flow graph, nested up to 4 levels.

We used our transition invariant generator to compute a transition invariant for the SVD program. Proving the transition invariant to be strong enough required testing termination of 219 LASW programs.

We applied our implementation of the algorithm on Figure 1, which was done in SICStus Prolog [Lab01] using the built-in constraint solver for linear arithmetic [Hol95]. Proving termination required 800 ms on a 2.6 GHz Xeon computer running Linux, which is in average 3.6 ms per each LASW program.

5 Related Work

The verification of termination and other liveness properties of programs requires the discovery of *invariants* as well as of *ranking functions* to prove the termination of program loops. Here, we relate our work not to methods for the automated discovery of invariants (see e.g. [Kar76,CH78,BBM97]), but to the more closely related topic of methods for the automated synthesis of ranking functions, a topic that has received increasing attention in the last years [GCGL02,CS01, DGG00,Mes96,MN01,CS02,SG91].

As a first general remark, a major difference between our work and all the others lies in the fact that we obtain a completeness result.

A heuristic-based approach for discovery of ranking functions is described in [DGG00]. It inspects the program source code for ranking function candidates. This method restricted to programs where the ranking function is exhibited already in the source code.

The algorithm in [CS01] extracts a linear ranking function of an unnested program loop by manipulating polyhedral cones; these represent the transition relation of the loop and the loop invariant. Their approach depends on the strength of the invariant generator, which they call in a subroutine to propose bounded linear arithmetic expression. The algorithm requires exponential space in the worst case. A generalization of that algorithm described in [CS02] for programs with complex control structures detects linear ranking functions for strongly connected components in the control-flow graph of more general programs. In both cases the algorithm is restricted to bounded nondeterminism. Moreover, it cannot handle loops with non-monotonic decrease, such as in `while (x>=0) {x=x+1; x=x-1;}`.

The method for discovery of nonnegative linear combinations of bound argument sizes for proving termination of logic programs in [SG91] relies on automatically inferred inter-argument constraints. The duality theory of linear programming is applied to discover combinations that decrease during top-down execution of recursive rules; the determined combinations are bounded from below since argument sizes are always positive. This method was applied for inferring termination of constraint logic programs [Mes96], and in systems for inferring termination of logic programs [MN01,GCGL02]. Carried over into the context of imperative program loops, the inference of inter-argument constraints corresponds to calls to the invariant generator, as in [CS01]; the same restrictions as mentioned above apply.

6 Conclusion

We have presented the to our knowledge first complete algorithm for the synthesis of linear ranking functions for a small but natural and well-motivated class of programs, namely, unnested program loops built using linear arithmetic expressions (LASW programs). The method is guaranteed to find a linear ranking function, and therefore to prove termination, if a linear ranking function exists. The existence of a linear ranking function for an LASW program is equivalent to the satisfiability of the system of linear inequalities derived from the program.

The termination check for LASW programs is a subroutine in the automated method for the verification of termination and other liveness properties of general-purpose programs via the computation of transition invariants [PR03].

We have implemented the proposed algorithm using an efficient implementation of a solver for linear programming over rationals [Hol95]. We applied our implementation to prove termination of a singular value decomposition program, which required termination proofs for 219 LASW programs. This and other experiments indicate the practical potention of the algorithm.

Considering future work, we would like to find a characterization of LASW programs that do always have linear ranking functions, i.e., for which our algorithm decides termination. Another direction of work is to handle unnested program loops built using expressions other than linear arithmetic.

Acknowledgments. We thank Bernd Finkbeiner, Konstantin Korovin, and Uwe Waldmann for comments on this paper. We thank Ramesh Kumar for his help with the SVD example.

References

[BBM97] Nikolaj Bjørner, Anca Browne, and Zohar Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, 1997.

[BS99] Olaf Burkart and Bernhard Steffen. Model checking the full modal mu-calculus for infinite sequential processes. *Theoretical Computer Science*, 221:251–270, 1999.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. of POPL 1978: Symp. on Principles of Programming Languages*, pages 84–97. ACM Press, 1978.

[CS01] Michael Colon and Henny Sipma. Synthesis of linear ranking functions. In Tiziana Margaria and Wang Yi, editors, *Proc. of TACAS 2001: Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *LNCS*, pages 67–81. Springer-Verlag, 2001.

[CS02] Michael Colon and Henny Sipma. Practical methods for proving program termination. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proc. of CAV 2002: Computer Aided Verification*, volume 2404 of *LNCS*, pages 442–454. Springer-Verlag, 2002.

[DGG00] Dennis Dams, Rob Gerth, and Orna Grumberg. A heuristic for the automatic generation of ranking functions. In *Workshop on Advances in Verification (WAVe'00)*, pages 1–8, 2000.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall Series in Automatic Computation. Prentice Hall, Englewood Cliffs, NJ, 1976.

[GCGL02] Samir Genaim, Michael Codish, John P. Gallagher, and Vitaly Lagoon. Combining norms to prove termination. In Agostino Cortesi, editor, *Proc. of VMCAI 2002: Verification, Model Checking, and Abstract Interpretation*, volume 2294 of *LNCS*, pages 126–138. Springer-Verlag, 2002.

[GVL96] Gene H. Golub and Charles F. Van Loan. *Matrix Computations; 3rd edition*. Johns Hopkins Univ Press, 3rd edition, 1996.

[Hol95] Christian Holzbaur. *OFAI clp(q,r) Manual, Edition 1.3.3*. Austrian Research Institute for Artificial Intelligence, Vienna, 1995. TR-95-09.

[Kar76] Michael Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.

[Lab01] The Intelligent Systems Laboratory. *SICStus Prolog User's Manual*. Swedish Institute of Computer Science, PO Box 1263 SE-164 29 Kista, Sweden, October 2001. Release 3.8.7.

[Mes96] Fred Mesnard. Inferring left-terminating classes of queries for constraint logic programs. In Michael J. Maher, editor, *Proc. of JICSLP 1996: Joint Int. Conf. and Symp. on Logic Programming*, pages 7–21. MIT Press, 1996.

[MN01] Fred Mesnard and Ulrich Neumerkel. Applying static analysis techniques for inferring termination conditions of logic programs. In Patrick Cousot, editor, *Proc. of SAS 2001: Symp. on Static Analysis*, volume 2126 of *LNCS*, pages 93–110. Springer-Verlag, 2001.

[PR03] Andreas Podelski and Andrey Rybalchenko. Software model checking of liveness properties via transition invariants. Technical report, Max-Plank-Institut für Informatik, 2003.

[PTVF92] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P . Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 1992.

[Sch86] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons Ltd., 1986.

[SG91] Kirack Sohn and Allen Van Gelder. Termination detection in logic programs using argument sizes. In *Proc. of PODS 1991: Symp. on Principles of Database Systems*, pages 216–226. ACM Press, 1991.

Symbolic Implementation of the Best Transformer

Thomas Reps[1], Mooly Sagiv[2], and Greta Yorsh[2]

[1] Comp. Sci. Dept., University of Wisconsin; reps@cs.wisc.edu
[2] School of Comp. Sci., Tel-Aviv University; {msagiv,gretay}@post.tau.ac.il

Abstract. This paper shows how to achieve, under certain conditions, abstract-interpretation algorithms that enjoy the best possible precision for a given abstraction. The key idea is a simple process of successive approximation that makes repeated calls to a decision procedure, and obtains the best abstract value for a set of concrete stores that are represented symbolically, using a logical formula.

1 Introduction

Abstract interpretation [6] is a well-established technique for automatically proving certain program properties. In abstract interpretation, sets of program stores are represented in a conservative manner by abstract values. Each program statement is given an interpretation over abstract values that is conservative with respect to its interpretation over corresponding sets of concrete stores; that is, the result of "executing" a statement must be an abstract value that describes a superset of the concrete stores that actually arise. This methodology guarantees that the results of abstract interpretation overapproximate the sets of concrete stores that actually arise at each point in the program.

In [7], it is shown that, under certain reasonable conditions, it is possible to give a *specification* of the most-precise abstract interpretation for a given abstract domain. For a Galois connection defined by abstraction function α and concretization function γ, the best abstract post operator for transition τ, denoted by $\mathrm{Post}^\sharp[\tau]$, can be expressed in terms of the concrete post operator for τ, $\mathrm{Post}[\tau]$, as follows:

$$\mathrm{Post}^\sharp[\tau] = \alpha \circ \mathrm{Post}[\tau] \circ \gamma. \tag{1}$$

This defines the limit of precision obtainable using a given abstraction. However, Eqn. (1) is non-constructive; it does not provide an *algorithm* for finding or applying $\mathrm{Post}^\sharp[\tau]$.

Graf and Saïdi [11] showed that decision procedures can be used to generate best abstract transformers for abstract domains that are fixed, finite, Cartesian products of Boolean values. (The use of such domains is known as *predicate abstraction*; predicate abstraction is also used in SLAM [2] and other systems [8,12].) The work presented in this paper shows how some of the benefits enjoyed by applications that use the predicate-abstraction approach can also be enjoyed by applications that use abstract domains other than predicate-abstraction domains. In particular, this paper's results apply to arbitrary finite-height abstract domains, not just to Cartesian products of Booleans. For example, it applies to the abstract domains used for constant propagation and common-subexpression elimination [14]. When applied to a predicate-abstraction domain, the method has the same worst-case complexity as the Graf-Saïdi method.

To understand where the difficulties lie, consider how they are addressed in predicate abstraction. In general, the result of applying γ to an abstract value l is an infinite set

B. Steffen and G. Levi (Eds.): VMCAI 2004, LNCS 2937, pp. 252–266, 2004.

of concrete stores; Graf and Saïdi sidestep this difficulty by performing γ symbolically, expressing the result of $\gamma(l)$ as a formula φ. They then introduce a function that, in effect, is the composition of α and $\mathrm{Post}[\tau]$: it applies $\mathrm{Post}[\tau]$ to φ and maps the result back to the abstract domain. In other words, Eqn. (1) is recast using two functions that work at the symbolic level, $\widehat{\gamma}$ and $\widehat{\alpha\mathrm{Post}}[\tau]$,[1] such that $\widehat{\alpha\mathrm{Post}}[\tau] \circ \widehat{\gamma} = \alpha \circ \mathrm{Post}[\tau] \circ \gamma$.

To provide insight on what opportunities exist as we move from predicate-abstraction domains to the more general class of finite-height lattices, we first address a simpler problem than $\widehat{\alpha\mathrm{Post}}[\tau]$, namely,

> How can $\widehat{\alpha}$ be implemented? That is, how can one identify the most-precise abstract value of a given abstract domain that overapproximates a set of concrete stores that are represented symbolically?

We then employ the basic idea used in $\widehat{\alpha}$ to implement our own version of $\widehat{\alpha\mathrm{Post}}[\tau]$.

The contributions of the paper can be summarized as follows:

- The paper shows how some of the benefits enjoyed by predicate abstraction can be extended to arbitrary finite-height abstract domains. In particular, we describe methods for each of the operations needed to carry out abstract interpretation.
- With some logics, the result of applying $\mathrm{Post}[\tau]$ to a given set of concrete stores (represented symbolically) can also be expressed symbolically, as a formula ϕ'. In this case, we can proceed by computing $\widehat{\alpha}(\phi')$. For other logics, however, ϕ' cannot be expressed symbolically without passing to a more powerful logic. For instance,
  - If sets of concrete stores are represented with quantifier-free first-order logic, it may require quantified first-order logic to express $\mathrm{Post}[\tau]$.
  - If sets of concrete stores are represented with a decidable subset of first-order logic, it may require second-order logic to express $\mathrm{Post}[\tau]$.

  In such situations, the procedure that we give to compute $\widehat{\alpha\mathrm{Post}}[\tau]$ provides a way to compute the best transformer while staying within the original logic.

The remainder of the paper is organized as follows: Sect. 2 motivates the work by presenting an $\widehat{\alpha}$ procedure for a specific finite-height lattice. Sect. 3 introduces terminology and notation. Sect. 4 presents the general treatment of $\widehat{\alpha}$ procedures for finite-height lattices. Sect. 5 discusses symbolic techniques for implementing transfer functions (i.e., $\widehat{\alpha\mathrm{Post}}[\tau]$). Sect. 6 makes some additional observations about the work. Sect. 7 discusses related work.

2 Motivating Examples

This section presents several examples to motivate the work. The treatment here is at a semi-formal level; a more formal treatment is given in later sections. (This section assumes a certain amount of background on abstract interpretation; some readers may find it helpful to consult Sect. 3 before reading this section.)

The example concerns a simple concrete domain: let Var denote the set of variables in the program being analyzed; the concrete domain is $2^{Var \to \mathcal{Z}}$.

[1] We use the diacritic $\widehat{\ }$ on a symbol to indicate an operation that either produces or operates on a symbolic representation of a set of concrete stores.

Predicate Abstraction. A predicate-abstraction domain $\mathcal{PA}[\mathcal{B}]$ is based on a set $\mathcal{B}$ of predicate names, each of which has an associated defining formula: $\mathcal{B} = \{B_j \stackrel{\text{def}}{=} \varphi_j \mid 1 \leq j \leq k\}$. Each value in $\mathcal{PA}[\mathcal{B}]$ is a set of possibly negated symbols drawn from $\mathcal{B}$, where each symbol B_j is either present in positive or negative form (but not both), or absent entirely. For instance, with $\mathcal{B} = \{B_1 \stackrel{\text{def}}{=} \varphi_1, B_2 \stackrel{\text{def}}{=} \varphi_2, B_3 \stackrel{\text{def}}{=} \varphi_3\}$, values in $\mathcal{PA}[\mathcal{B}]$ include $\{\neg B_1, B_2, \neg B_3\}$, $\{B_1, B_2\}$, $\{\neg B_3\}$, and $\emptyset$.

We will use a predicate-abstraction domain in which there is a Boolean predicate $B \stackrel{\text{def}}{=} (x = c)$ for each $\mathtt{x} \in \mathit{Var}$ and each distinct constant c that appears in the program. For instance, if the program is

$$\begin{array}{l} \mathtt{y := 3} \\ \mathtt{x := 4 * y + 1} \end{array} \tag{2}$$

the predicate-abstraction domain is based on the predicate set $\{B_1 \stackrel{\text{def}}{=} (y = 1)$, $B_2 \stackrel{\text{def}}{=} (y = 3)$, $B_3 \stackrel{\text{def}}{=} (y = 4)$, $B_4 \stackrel{\text{def}}{=} (x = 1)$, $B_5 \stackrel{\text{def}}{=} (x = 3)$, $B_6 \stackrel{\text{def}}{=} (x = 4)\}$. Note that this domain does not provide an exact representation of the final state that arises, $[x \mapsto 13, y \mapsto 3]$. The best that can be done is to use the abstract value $\{\neg B_1, B_2, \neg B_3, \neg B_4, \neg B_5, \neg B_6\}$, which provides limited information about the value of x.

Our choice of predicate-abstraction domain $\mathcal{PA}[\{B_1, B_2, B_3, B_4, B_5, B_6\}]$ was made solely for the sake of simplicity. With a different choice of predicates, we could have retained a greater or lesser amount of information about the value of x in the state after program (2); however, there would always be some program that gives rise to a state in which information is lost.

The α Function for Predicate-Abstraction Domains. One of the virtues of the predicate-abstraction method is that it provides a procedure to obtain a most-precise abstract value, given (a specification of) a set of concrete stores as a logical formula ψ [11]. We will call this procedure $\widehat{\alpha}_{\mathrm{PA}}$; it relies on the aid of a decision procedure, and can be defined as follows:

$$\widehat{\alpha}_{\mathrm{PA}}(\psi) = \{B_j \mid \psi \Rightarrow \varphi_j \text{ is valid}\} \cup \{\neg B_j \mid \psi \Rightarrow \neg\varphi_j \text{ is valid}\} \tag{3}$$

For instance, suppose that ψ is the formula $(y = 3) \wedge (x = 4 * y + 1)$, which captures the final state of program (2). For $\widehat{\alpha}_{\mathrm{PA}}((y = 3) \wedge (x = 4 * y + 1))$ to produce the answer $\{\neg B_1, B_2, \neg B_3, \neg B_4, \neg B_5, \neg B_6\}$, the decision procedure must demonstrate that the following formulas are valid:

$$\begin{array}{ll} (y = 3) \wedge (x = 4 * y + 1) \Rightarrow \neg(y = 1) & (y = 3) \wedge (x = 4 * y + 1) \Rightarrow \neg(x = 1) \\ (y = 3) \wedge (x = 4 * y + 1) \Rightarrow \quad (y = 3) & (y = 3) \wedge (x = 4 * y + 1) \Rightarrow \neg(x = 3) \\ (y = 3) \wedge (x = 4 * y + 1) \Rightarrow \neg(y = 4) & (y = 3) \wedge (x = 4 * y + 1) \Rightarrow \neg(x = 4) \end{array}$$

Going Beyond Predicate Abstraction. We now show that the ability to implement the α function of a Galois connection between a concrete and abstract domain is not limited to predicate-abstraction domains. In particular, we will demonstrate this for the abstract

domain used in the constant-propagation problem: $(Var \to \mathcal{Z}^\top)_\bot$. The abstract value $\bot$ represents $\emptyset$; an abstract value such as $[\mathtt{x} \mapsto 0, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto 0]$ represents all concrete stores in which program variables x and z are both mapped to 0.[2]

The procedure to implement $\widehat{\alpha}$ for the constant-propagation domain, which we call $\widehat{\alpha}_{\mathrm{CP}}$, is actually an instance of a general procedure for implementing $\widehat{\alpha}$ functions that applies to a family of Galois connections. It is presented in Fig. 1; $\widehat{\alpha}_{\mathrm{CP}}$ is the instance of this procedure in which the return type L is $(Var \to \mathcal{Z}^\top)_\bot$, and "structure" in line [5] means "concrete store".

```
[1]  L α̂(formula ψ) {
[2]    ans := ⊥
[3]    φ := ψ
[4]    while (φ is satisfiable) {
[5]      Select a structure S such that S ⊨ φ
[6]      ans := ans ⊔ β(S)
[7]      φ := φ ∧ ¬γ̂(ans)
[8]    }
[9]    return ans
[10] }
```

Fig. 1. An algorithm to obtain, with the aid of a decision procedure, a most-precise abstract value that overapproximates a set of concrete stores. In Sect. 2, the return type L is $(Var \to \mathcal{Z}^\top)_\bot$, and "structure" in line [5] means "concrete store".

As with procedure $\widehat{\alpha}_{\mathrm{PA}}$, $\widehat{\alpha}_{\mathrm{CP}}$ is permitted to make calls to a decision procedure (see line [5] of Fig. 1). We make one assumption that goes beyond what is assumed in predicate abstraction, namely, we assume that the decision procedure is a satisfiability checker that is capable of returning a satisfying assignment, or, equivalently, that it is a validity checker that returns a counterexample. (In the latter case, the counterexample obtained by calling `ProveValid(`$\neg\varphi$`)` is a suitable satisfying assignment.)

The other operations used in procedure $\widehat{\alpha}_{\mathrm{CP}}$ are β, $\sqcup$, and $\widehat{\gamma}$:

- The concrete and abstract domains are related by a Galois connection defined by a representation function β that maps a concrete store $S \in Var \to \mathcal{Z}$ to an abstract value $\beta(S) \in (Var \to \mathcal{Z}^\top)_\bot$. For instance, β maps the concrete store $[x \mapsto 13, y \mapsto 3]$ to the abstract value $[\mathtt{x} \mapsto 13, \mathtt{y} \mapsto 3]$.
- $\sqcup$ is the join operation in $(Var \to \mathcal{Z}^\top)_\bot$. For instance,

$$[\mathtt{x} \mapsto 0, \mathtt{y} \mapsto 43, \mathtt{z} \mapsto 0] \sqcup [\mathtt{x} \mapsto 0, \mathtt{y} \mapsto 46, \mathtt{z} \mapsto 0] = [\mathtt{x} \mapsto 0, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto 0].$$

[2] We write abstract values in Courier typeface (e.g., $[\mathtt{x} \mapsto 0, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto 0]$), and concrete stores in Roman typeface (e.g., $[x \mapsto 0, y \mapsto 43, z \mapsto 0]$).

– There is an operation $\widehat{\gamma}$ that maps an abstract value l to a formula $\widehat{\gamma}(l)$ such that l and $\widehat{\gamma}(l)$ represent the same set of concrete stores. For instance, we have

$$\widehat{\gamma}([\mathtt{x} \mapsto 0, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto 0]) = (x = 0) \wedge (z = 0).$$

The resulting formula contains no term involving y because $\mathtt{y} \mapsto \top$ does not place any restrictions on the value of y.

Operation $\widehat{\gamma}$ permits the concretization of an abstract store to be represented symbolically, using a logical formula. This allows sets of concrete stores to be manipulated symbolically, via operations on formulas.

To see how $\widehat{\alpha}_{\mathrm{CP}}$ works, consider the program

$$\begin{array}{l} \mathtt{z} := 0 \\ \mathtt{x} := \mathtt{y} * \mathtt{z} \end{array} \tag{4}$$

and suppose that ψ is the formula $(z = 0) \wedge (x = y * z)$, which captures the final state of program (4). The following sequence of operations would be performed during the invocation of $\widehat{\alpha}_{\mathrm{CP}}((z = 0) \wedge (x = y * z))$:

Initialization:	$\mathtt{ans} := \bot$	
	$\varphi := (z = 0) \wedge (x = y * z)$	
Iteration 1:	$S := [x \mapsto 0, y \mapsto 43, z \mapsto 0]$	// Some satisfying concrete store
	$\mathtt{ans} := \bot \sqcup \beta([x \mapsto 0, y \mapsto 43, z \mapsto 0])$	
	$= [\mathtt{x} \mapsto 0, \mathtt{y} \mapsto 43, \mathtt{z} \mapsto 0]$	
	$\widehat{\gamma}(\mathtt{ans}) = (x = 0) \wedge (y = 43) \wedge (z = 0)$	
	$\varphi := (z = 0) \wedge (x = y * z) \wedge \neg((x = 0) \wedge (y = 43) \wedge (z = 0))$	
	$= (z = 0) \wedge (x = y * z) \wedge ((x \neq 0) \vee (y \neq 43) \vee (z \neq 0))$	
	$= (z = 0) \wedge (x = y * z) \wedge (y \neq 43)$	
Iteration 2:	$S := [x \mapsto 0, y \mapsto 46, z \mapsto 0]$	// Some satisfying concrete store
	$\mathtt{ans} := [\mathtt{x} \mapsto 0, \mathtt{y} \mapsto 43, \mathtt{z} \mapsto 0] \sqcup \beta([x \mapsto 0, y \mapsto 46, z \mapsto 0])$	
	$= [\mathtt{x} \mapsto 0, \mathtt{y} \mapsto 43, \mathtt{z} \mapsto 0] \sqcup [\mathtt{x} \mapsto 0, \mathtt{y} \mapsto 46, \mathtt{z} \mapsto 0]$	
	$= [\mathtt{x} \mapsto 0, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto 0]$	
	$\widehat{\gamma}(\mathtt{ans}) = (x = 0) \wedge (z = 0)$	
	$\varphi := (z = 0) \wedge (x = y * z) \wedge (y \neq 43) \wedge ((x \neq 0) \vee (z \neq 0))$	
	$=$ **ff**	
Iteration 3:	φ is unsatisfiable	
Return value:	$[\mathtt{x} \mapsto 0, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto 0]$	

At this point the loop terminates, and $\widehat{\alpha}_{\mathrm{CP}}$ returns the abstract value $[\mathtt{x} \mapsto 0, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto 0]$. In effect, $\widehat{\alpha}_{\mathrm{CP}}$ has automatically discovered that in the abstract world the best treatment of the multiplication operator is for it to be non-strict in $\top$. That is, 0 is a multiplicative annihilator that supersedes $\top$: $0 = \top * 0$.

In general, $\widehat{\alpha}(\psi)$ carries out a process of successive approximation, making repeated calls to a decision procedure. Initially, φ is set to ψ and ans is set to $\bot$. On each iteration of the loop in $\widehat{\alpha}$, the value of ans becomes a better approximation of the desired answer, and the value of φ describes a smaller set of concrete stores, namely, those stores described by ψ that are not, as yet, covered by ans. For instance, at line [7] of Fig. 1 during Iteration 1 of the second example of $\widehat{\alpha}_{\mathrm{CP}}(\psi)$, ans has the value $[\mathtt{x} \mapsto 0, \mathtt{y} \mapsto 43, \mathtt{z} \mapsto 0]$, and the update to φ, $\varphi := \varphi \wedge \neg\widehat{\gamma}(\mathtt{ans})$, sets φ to $(z = 0) \wedge (x = y * z) \wedge (y \neq 43)$. Thus, φ describes exactly the stores that are described by ψ, but are not, as yet, covered by ans.

Each time around the loop, $\widehat{\alpha}$ selects a concrete store S such that $S \models \varphi$. Then $\widehat{\alpha}$ uses β and $\sqcup$ to perform what can be viewed as a "generalization" operation: β converts concrete store S into an abstract store; the current value of `ans` is augmented with $\beta(S)$ using $\sqcup$. For instance, at line [6] of Fig. 1 during Iteration 2 of the second example of $\widehat{\alpha}_{\mathrm{CP}}(\psi)$, `ans`'s value is changed from $[\mathtt{x} \mapsto 0, \mathtt{y} \mapsto 43, \mathtt{z} \mapsto 0]$ to $[\mathtt{x} \mapsto 0, \mathtt{y} \mapsto 43, \mathtt{z} \mapsto 0] \sqcup \beta([x \mapsto 0, y \mapsto 46, z \mapsto 0]) = [\mathtt{x} \mapsto 0, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto 0]$. In other words, the generalization from two possible values for `y`, `43` and `46`, is $\top$, which indicates that `y` may not be a constant at the end of the program.

Fig. 2 presents a sequence of diagrams that illustrate schematically algorithm $\widehat{\alpha}$ from Fig. 1.

3 Terminology and Notation

For us, concrete stores are *logical structures*. The advantage of adopting this outlook is that it allows potentially infinite sets of concrete stores to be represented using formulas.

Definition 1. *Let $P = \{p_1, \ldots, p_m\}$ be a finite set of predicate symbols, each with a fixed arity; let P_i denote the set of predicate symbols with arity i. Let $C = \{c_1, \ldots, c_n\}$ be a finite set of constant symbols. Let $F = \{f_1, f_2, \ldots, f_p\}$ be a finite set of function symbols each with a fixed arity; let F_i denote the set of function symbols with arity i. A* **logical structure** *over* **vocabulary** *$V = \langle P, C, F\rangle$ is a tuple $S = \langle U, \iota_p, \iota_c, \iota_f\rangle$ in which*

- *U is a (possibly infinite) set of individuals.*
- *ι_p is the interpretation of predicate symbols, i.e., for every predicate symbol $p \in P_i$, $\iota_p(p) \subseteq U^i$ denotes the set of i-tuples for which p holds.*
- *ι_c is the interpretation of constant symbols, i.e., for every constant symbol $c \in C$, $\iota_c(c) \in U$ denotes the individual associated with c.*
- *ι_f is the interpretation of function symbols, i.e., for every function symbol $f \in F_i$, $\iota_f(f) \colon U^i \to U$ maps i-tuples into an individual.*

Typically, some subset of the predicate symbols, constant symbols, and function symbols have an interpretation that is fixed in advance; this defines a family of intended models. We denote the (infinite) set of structures over V, where the interpretations of $I \subseteq V$ are fixed in advance, by $ConcreteStruct[V, I]$.

Example 1. In Sect. 2, we considered concrete stores to be members of $Var \to \mathcal{Z}$. This is a common way to define concrete stores; however, in the remainder of the paper concrete stores are identified with logical structures. A store in which program variables are bound to integer values is a logical structure $\langle \mathcal{Z}, \emptyset, \iota_{Var}, \emptyset\rangle$ over vocabulary $\langle IntPreds, Var \cup IntConsts, IntFuncs\rangle$, where ι_{Var} is a mapping of program variables to integers, and the symbols in $IntPreds = \{<, \leq, =, \neq, \geq, >, \ldots\}$, $IntConsts = \{0, -1, 1, -2, 2, \ldots\}$, and $IntFuncs = \{+, -, *, /, \ldots\}$ have their usual meanings. For instance, an example concrete store for a program in which $Var = \{x, y, z\}$ is

$$\langle \mathcal{Z}, \emptyset, [x \mapsto 0, y \mapsto 2, z \mapsto 0], \emptyset\rangle. \tag{5}$$

Henceforth, we abbreviate a store such as (5) by $\iota_c = [x \mapsto 0, y \mapsto 2, z \mapsto 0]$.

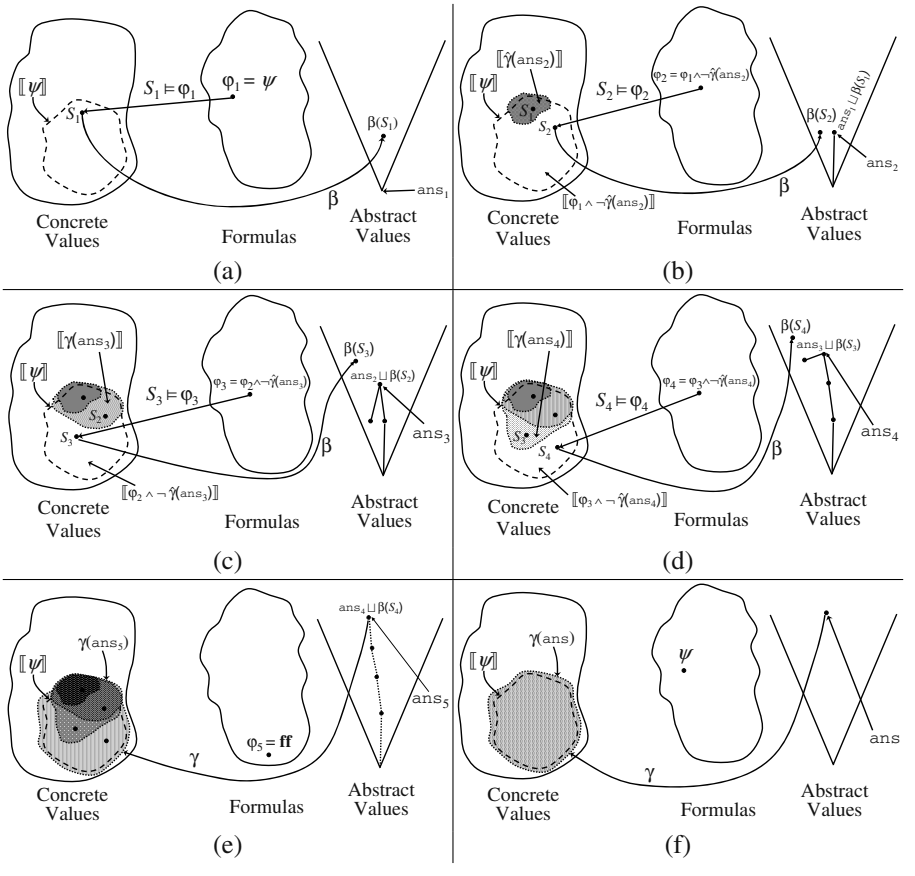


Fig. 2. Schematic diagrams that illustrate the process carried out by algorithm $\widehat{\alpha}(\psi)$ from Fig. 1; φ_i, S_i, and $\texttt{ans}_i$ denote the values of φ, S, and ans during the i^{th} iteration. (a) Initially, φ_1 is set to ψ and $\texttt{ans}_1$ is set to $\bot$; S_1 is a structure such that $S_1 \models \varphi_1$. (b) $\texttt{ans}_2$ is set to $\texttt{ans}_1 \sqcup \beta(S_1) = \beta(S_1)$; φ_2 is set to $\varphi_1 \wedge \neg\widehat{\gamma}(\texttt{ans}_2)$; S_2 is a structure such that $S_2 \models \varphi_2$. Note that S_2 belongs to $[\![\varphi_2]\!] = [\![\varphi_1 \wedge \neg\widehat{\gamma}(\texttt{ans}_2)]\!]$. (c) $\texttt{ans}_3$ is set to $\texttt{ans}_2 \sqcup \beta(S_2)$; φ_3 is set to $\varphi_2 \wedge \neg\widehat{\gamma}(\texttt{ans}_3)$; S_3 is a structure such that $S_3 \models \varphi_3$. (d) $\texttt{ans}_4$ is set to $\texttt{ans}_3 \sqcup \beta(S_3)$; φ_4 is set to $\varphi_3 \wedge \neg\widehat{\gamma}(\texttt{ans}_4)$; S_4 is a structure such that $S_4 \models \varphi_4$. (e) $\texttt{ans}_5$ is set to $\texttt{ans}_4 \sqcup \beta(S_4)$; φ_5 is set to $\varphi_4 \wedge \neg\widehat{\gamma}(\texttt{ans}_5)$. In the case portrayed here, the loop terminates at this point because $\varphi_5 = \mathbf{ff}$. The desired answer is held in $\texttt{ans}_5$. (f) $\widehat{\alpha}(\psi)$ obtains the most-precise abstract value ans that overapproximates $[\![\psi]\!]$.

To manipulate sets of structures symbolically, we use formulas of first-order logic with equality. If S is a logical structure and φ is a closed formula, the notation $S \models \varphi$ means that S satisfies φ according to the standard Tarskian semantics for first-order logic (e.g., see [10]). We use $[\![\varphi]\!]$ to denote the set of concrete structures that satisfy φ: $[\![\varphi]\!] = \{S \mid S \in \mathit{ConcreteStruct}[V, I], S \models \varphi\}$.

Example 2.

$$[\![(x = 0) \wedge (z = 0)]\!] = \left\{ \begin{array}{l} \iota_c = [x \mapsto 0, y \mapsto 0, z \mapsto 0],\ \iota_c = [x \mapsto 0, y \mapsto 1, z \mapsto 0], \\ \iota_c = [x \mapsto 0, y \mapsto 2, z \mapsto 0],\ \ldots \end{array} \right\}$$

Definition 2. *A* **complete join semilattice** $L = \langle L, \sqsubseteq, \bigsqcup, \bot \rangle$ *is a partially ordered set with partial order* $\sqsubseteq$*, such that for every subset* X *of* L*,* L *contains a least upper bound (or* **join***), denoted by* $\bigsqcup X$*.*

The minimal element $\bot \in L$ *is* $\bigsqcup \emptyset$*. We use* $x \sqcup y$ *as a shorthand for* $\bigsqcup\{x, y\}$*. We write* $x \sqsubset y$ *when* $x \sqsubseteq y$ *and* $x \neq y$*.*

The powerset of concrete stores $2^{ConcreteStruct[V,I]}$ is a complete join semilattice, where (i) $X \sqsubseteq Y$ iff $X \subseteq Y$, (ii) $\bot = \emptyset$, and (iii) $\bigsqcup = \bigcup$.

Definition 3. *Let* $L = \langle L, \sqsubseteq, \bigsqcup, \bot \rangle$ *be a complete join semilattice. A* **strictly increasing chain in** L *is a sequence of values* $l_1, l_2, \ldots$*, such that* $l_i \sqsubset l_{i+1}$*. We say that* L **has finite height** *if every strictly increasing chain is finite.*

We now define an abstract domain by means of a representation function [18].

Definition 4. *Given a complete join semilattice* $L = \langle L, \sqsubseteq, \bigsqcup, \bot \rangle$ *and a* **representation function** $\beta\colon ConcreteStruct[V, I] \to L$ *such that for all* $S \in ConcreteStruct[V, I]$ $\beta(S) \neq \bot$*, a Galois connection* $2^{ConcreteStruct[V,I]} \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} L$ *is defined by extending* β *pointwise, i.e., for* $XS \subseteq ConcreteStruct[V, I]$ *and* $l \in L$*,*

$$\alpha(XS) = \bigsqcup_{S \in XS} \beta(S) \qquad \gamma(l) = \{S \mid S \in ConcreteStruct[V, I], \beta(S) \sqsubseteq l\}$$

It is straightforward to show that this defines a Galois connection, i.e., (i) α *and* γ *are monotonic, (ii)* α *distributes over* $\cup$*, (iii)* $XS \subseteq \gamma(\alpha(XS))$*, and (iv)* $\alpha(\gamma(l)) \sqsubseteq l$*.*

We say that l **overapproximates** *a set of concrete stores* XS *if* $\gamma(l) \supseteq XS$*. It is straightforward to show that* $\alpha(XS)$ *is the most-precise (i.e., least) abstract value that overapproximates* XS*.*

Example 3. In our examples, the abstract domain will continue to be the one introduced in Sect. 2, namely, $(Var \to \mathcal{Z}^\top)_\bot$. As we saw in Sect. 2, β maps a concrete store like $\iota_c = [x \mapsto 0, y \mapsto 2, z \mapsto 0]$ to an abstract value $[\mathtt{x} \mapsto \mathtt{0}, \mathtt{y} \mapsto \mathtt{2}, \mathtt{z} \mapsto \mathtt{0}]$. Thus,

$$\begin{aligned} \alpha\left(\left\{ \begin{array}{l} \iota_c = [x \mapsto 0, y \mapsto 0, z \mapsto 0], \\ \iota_c = [x \mapsto 0, y \mapsto 2, z \mapsto 0] \end{array} \right\}\right) &= \begin{array}{r} \beta(\iota_c = [x \mapsto 0, y \mapsto 0, z \mapsto 0]) \\ \sqcup\, \beta(\iota_c = [x \mapsto 0, y \mapsto 2, z \mapsto 0]) \end{array} \\ &= \begin{array}{r} [\mathtt{x} \mapsto \mathtt{0}, \mathtt{y} \mapsto \mathtt{0}, \mathtt{z} \mapsto \mathtt{0}] \\ \sqcup\, [\mathtt{x} \mapsto \mathtt{0}, \mathtt{y} \mapsto \mathtt{2}, \mathtt{z} \mapsto \mathtt{0}] \end{array} \\ &= [\mathtt{x} \mapsto \mathtt{0}, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto \mathtt{0}]. \end{aligned}$$

Suppose that abstract value l is $[\mathtt{x} \mapsto \mathtt{0}, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto \mathtt{0}]$. Because $\mathtt{y} \mapsto \top$ does not place any restrictions on the value of $\mathtt{y}$, we have

$$\gamma(l) = \left\{ \begin{array}{l} \iota_c = [x \mapsto 0, y \mapsto 0, z \mapsto 0],\ \iota_c = [x \mapsto 0, y \mapsto 1, z \mapsto 0], \\ \iota_c = [x \mapsto 0, y \mapsto 2, z \mapsto 0],\ \ldots \end{array} \right\}$$

4 Symbolic Implementation of the α Function

This section presents a general framework for implementing α functions of Galois connections using procedure $\widehat{\alpha}$ from Fig. 1. $\widehat{\alpha}(\psi)$ finds the most-precise abstract value in a finite-height lattice, given a specification of a set of concrete stores as a logical formula ψ. $\widehat{\alpha}$ represents sets of concrete stores symbolically, using formulas, and invokes a decision procedure on each iteration.

The assumptions of the framework are rather minimal:

- The concrete domain is the power set of $ConcreteStruct[V, I]$.
- The concrete and abstract domains are related by a Galois connection defined by a representation function β that maps a structure $S \in ConcreteStruct[V, I]$ to an abstract value $\beta(S)$.
- It is possible to take the join of two abstract values.
- There is an operation $\widehat{\gamma}$ that maps an abstract value l to a formula $\widehat{\gamma}(l)$ such that

$$[\![\widehat{\gamma}(l)]\!] = \gamma(l). \tag{6}$$

Operation $\widehat{\gamma}$ permits the concretization of an abstract value to be represented symbolically, using a logical formula, which allows sets of concrete stores to be manipulated symbolically, via operations on formulas. (In this paper, we use first-order logic; in general, however, other logics could be used.)

Example 4. As we saw in Sect. 2, because $\mathtt{y} \mapsto \top$ does not place any restrictions on the value of y, we have $\widehat{\gamma}([\mathtt{x} \mapsto \mathtt{0}, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto \mathtt{0}]) = (x = 0) \wedge (z = 0)$. From Exs. 2 and 3, we know that

$$\begin{aligned}[\![(x = 0) \wedge (z = 0)]\!] &= \left\{ \begin{array}{l} \iota_c = [x \mapsto 0, y \mapsto 0, z \mapsto 0],\ \iota_c = [x \mapsto 0, y \mapsto 1, z \mapsto 0], \\ \iota_c = [x \mapsto 0, y \mapsto 2, z \mapsto 0],\ \ldots \end{array} \right\} \\ &= \gamma([\mathtt{x} \mapsto \mathtt{0}, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto \mathtt{0}]),\end{aligned}$$

and thus Eqn. (6) is satisfied. For $l \in (Var \to \mathcal{Z}^\top)_\perp$, $\widehat{\gamma}(l)$ is defined as follows:

$$\widehat{\gamma}(l) = \begin{cases} \mathbf{ff} & \text{if } l = \perp \\ \bigwedge\limits_{\substack{v \in Var, \\ l(v) \neq \top}} (v = l(v)) & \text{otherwise} \end{cases}$$

Specification of Alpha. Procedure $\widehat{\alpha}$ is to implement α, given a specification of a set of concrete stores as a logical formula ψ. Therefore, $\widehat{\alpha}$ must have the property that for all ψ, $\widehat{\alpha}(\psi) = \alpha([\![\psi]\!])$.

Note that a logical formula ψ represents the set of concrete stores $[\![\psi]\!]$; thus, $\alpha([\![\psi]\!])$ (and hence $\widehat{\alpha}(\psi)$, as well) is the most-precise abstract value that overapproximates the set of concrete stores represented symbolically by ψ.

Implementation of Alpha. Procedure $\widehat{\alpha}$ is given in Fig. 1.

Example 5. A trace of a call on $\widehat{\alpha}$ for the constant-propagation domain $(Var \to \mathcal{Z}^\top)_\perp$ was presented in Sect. 2. In generalizing the idea from Sect. 2, concrete stores have been identified with logical structures, so instead of writing, e.g., $S := [x \mapsto 0, y \mapsto 43, z \mapsto 0]$, we would now write $S := \iota_c = [x \mapsto 0, y \mapsto 43, z \mapsto 0]$.

Theorem 1. *Suppose that the abstract domain has finite height of at most h. Given input ψ, $\widehat{\alpha}(\psi)$ has the following properties:*

(i) *The loop on lines [4]–[8] in procedure $\widehat{\alpha}$ is executed at most h times.*
(ii) $\widehat{\alpha}(\psi) = \alpha([\![\psi]\!])$ *(i.e., $\widehat{\alpha}(\psi)$ computes the most-precise abstract value that overapproximates the set of concrete stores represented symbolically by ψ).*

5 Symbolic Implementation of Transfer Functions

5.1 Transfer Functions for Statements

If Q is a set of predicate, constant, or function symbols, let Q' denote the same set of symbols, but with a $'$ attached to each symbol (i.e., $q \in Q$ iff $q' \in Q'$).

The interpretation of statements involves the specification of transition relations using formulas. Such formulas will be over a "double vocabulary" $V \cup V' = \langle P \cup P', C \cup C', F \cup F' \rangle$, where unprimed symbols will be referred to as *present-state* symbols, and primed symbols as *next-state* symbols.[3] The satisfaction relation for a two-vocabulary formula τ will be written as $\langle S, S' \rangle \models \tau$, where S and S' are structures over vocabularies $V = \langle P, C, F \rangle$ and $V' = \langle P', C', F' \rangle$, respectively; $\langle S, S' \rangle$ is called a *two-vocabulary structure*.

Example 6. The formula that expresses the semantics of an assignment $\mathtt{x} := \mathtt{y} * \mathtt{z}$ with respect to stores over vocabulary $\langle IntPreds, Var \cup Var' \cup IntConsts, IntFuncs \rangle$, denoted by $\tau_{\mathtt{x:=y*z}}$, can be specified as $\tau_{\mathtt{x:=y*z}} \stackrel{\text{def}}{=} (x' = y * z) \wedge (y' = y) \wedge (z' = z)$.

For parallel form, we will also assume that we have two isomorphic abstract domains, L and L', and associated variants of β and $\widehat{\gamma}$

$$\beta\colon \mathit{ConcreteStruct}[V, I] \to L \qquad \beta'\colon \mathit{ConcreteStruct}[V', I] \to L'$$
$$\widehat{\gamma}\colon L \to \mathit{Formula}[V] \qquad \widehat{\gamma}'\colon L' \to \mathit{Formula}[V']$$

For the constant-propagation domain, this just means that a next-state abstract value produced by one transition, e.g., $[\mathtt{x}' \mapsto 0, \mathtt{y}' \mapsto \top, \mathtt{z}' \mapsto 0] \in L'$, can be identified as the present-state abstract value $[\mathtt{x} \mapsto 0, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto 0] \in L$ for the next transition.[4]

[3] For economy of notation, we will not duplicate the symbols $I \subseteq V$ whose interpretation is fixed in advance.

[4] Alternatively, we could have used a single abstract domain, L, and the definitions

$$\beta\colon \mathit{ConcreteStruct}[V, I] \to L \qquad \beta'\colon \mathit{ConcreteStruct}[V', I] \to L$$
$$\widehat{\gamma}\colon L \to \mathit{Formula}[V] \qquad \widehat{\gamma}'\colon L \to \mathit{Formula}[V']$$

The motivation for using two abstract domains is to eliminate a possible source of confusion in the examples. By using separate abstract domains L and L', primed symbols always distinguish next-state abstract values from present-state ones.

Specification. Given a formula τ for a statement's transition relation, the result of applying τ to a set of concrete stores XS is

$$\mathrm{Post}[\tau](XS) = \{S' \mid \text{exists } S \in XS \text{ such that } \langle S, S' \rangle \models \tau\}.$$

(Note that this is a set of structures over vocabulary V'.) $\widehat{\alpha\mathrm{Post}}[\tau](l)$ is to return the most-precise abstract value in L' that overapproximates $\mathrm{Post}[\tau](\gamma(l))$.

Implementation. $\widehat{\alpha\mathrm{Post}}[\tau](l)$ can be computed by the procedure presented in Fig. 3. After φ is initialized to $\widehat{\gamma}(l) \wedge \tau$ in line [3], $\widehat{\alpha\mathrm{Post}}[\tau]$ operates very much like $\widehat{\alpha}$, except that only abstractions of the S' structures are accumulated in variable `ans'` (see lines [5] and [6]). On each iteration of the loop in $\widehat{\alpha\mathrm{Post}}[\tau]$, the value of `ans'` becomes a better approximation of the desired answer, and the value of φ describes a smaller set of concrete stores, namely, those $V \cup V'$ stores that are described by $\widehat{\gamma}(l) \wedge \tau$, but whose range (i.e., projection on the next-state symbols) is not, as yet, covered by `ans'`.

```
[1]  L' αPost(two-vocabulary formula τ over V ∪ V', L l) {
[2]     ans' := ⊥'
[3]     φ := γ(l) ∧ τ
[4]     while (φ is satisfiable) {
[5]       Select a two-vocabulary structure ⟨S, S'⟩ s.t. ⟨S, S'⟩ ⊨ φ
[6]       ans' := ans' ⊔ β'(S')
[7]       φ := φ ∧ ¬γ'(ans')
[8]     }
[9]     return ans'
[10] }
```

Fig. 3. An algorithm that implements $\widehat{\alpha\mathrm{Post}}[\tau](l)$.

Example 7. Suppose that $l = [\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto 0]$, and the statement to be interpreted is $\mathtt{x} := \mathtt{y} * \mathtt{z}$. Then $\widehat{\gamma}(l)$ is the formula $(z = 0)$, and $\tau_{\mathtt{x}:=\mathtt{y}*\mathtt{z}}$ is the formula $(x' = y * z) \wedge (y' = y) \wedge (z' = z)$. Fig. 4 shows why we have

$$\widehat{\alpha\mathrm{Post}}[\tau_{\mathtt{x}:=\mathtt{y}*\mathtt{z}}]([\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto 0]) = [\mathtt{x}' \mapsto 0, \mathtt{y}' \mapsto \top, \mathtt{z}' \mapsto 0].$$

Theorem 2. *Suppose that the abstract domain has finite height of at most h. Given inputs τ and l, $\widehat{\alpha\mathrm{Post}}[\tau](l)$ has the following properties:*

(i) *The loop on lines [4]–[8] in procedure $\widehat{\alpha\mathrm{Post}}[\tau](l)$ is executed at most h times.*
(ii) $\widehat{\alpha\mathrm{Post}}[\tau](l) = \alpha(\mathrm{Post}[\tau](\gamma(l)))$ *(i.e., $\widehat{\alpha\mathrm{Post}}[\tau](l)$ computes the most-precise abstract value in L' that overapproximates $\mathrm{Post}[\tau](\gamma(l))$).*

The operator $\mathrm{Pre}[\tau]$ can be implemented using a procedure that is dual to Fig. 3.

$$
\begin{array}{lrl}
\text{Initialization:} & \mathtt{ans}' & := \bot' \\
& \varphi & := (z = 0) \wedge (x' = y * z) \wedge (y' = y) \wedge (z' = z) \\
\text{Iteration 1:} & \langle S, S' \rangle & := \iota_c = \begin{bmatrix} x \mapsto 5, y \mapsto 17, z \mapsto 0 \\ x' \mapsto 0, y' \mapsto 17, z' \mapsto 0 \end{bmatrix} \quad \text{// Some satisfying structure} \\
& \mathtt{ans}' & := [\mathtt{x}' \mapsto 0, \mathtt{y}' \mapsto 17, \mathtt{z}' \mapsto 0] \\
& \widehat{\gamma}'(\mathtt{ans}') & = (x' = 0) \wedge (y' = 17) \wedge (z' = 0) \\
& \varphi & := (z = 0) \wedge (x' = y * z) \wedge (y' = y) \wedge (z' = z) \\
& & \quad \wedge ((x' \neq 0) \vee (y' \neq 17) \vee (z' \neq 0)) \\
& & = (z = 0) \wedge (x' = y * z) \wedge (y' = y) \wedge (z' = z) \wedge (y' \neq 17) \\
\text{Iteration 2:} & \langle S, S' \rangle & := \iota_c = \begin{bmatrix} x \mapsto 12, y \mapsto 99, z \mapsto 0 \\ x' \mapsto 0, y' \mapsto 99, z' \mapsto 0 \end{bmatrix} \quad \text{// Some satisfying structure} \\
& \mathtt{ans}' & := [\mathtt{x}' \mapsto 0, \mathtt{y}' \mapsto 17, \mathtt{z}' \mapsto 0] \sqcup [\mathtt{x}' \mapsto 0, \mathtt{y}' \mapsto 99, \mathtt{z}' \mapsto 0] \\
& & = [\mathtt{x}' \mapsto 0, \mathtt{y}' \mapsto \top, \mathtt{z}' \mapsto 0] \\
& \widehat{\gamma}'(\mathtt{ans}') & = (x' = 0) \wedge (z' = 0) \\
& \varphi & := (z = 0) \wedge (x' = y * z) \wedge (y' = y) \wedge (z' = z) \wedge (y' \neq 17) \\
& & \quad \wedge ((x' \neq 0) \vee (z' \neq 0)) \\
& & = \mathbf{ff} \\
\text{Iteration 3:} & & \varphi \text{ is unsatisfiable} \\
\text{Return value:} & & [\mathtt{x}' \mapsto 0, \mathtt{y}' \mapsto \top, \mathtt{z}' \mapsto 0]
\end{array}
$$

Fig. 4. Operations performed during a call $\widehat{\alpha\mathrm{Post}}[\tau_{\mathtt{x}:=\mathtt{y}*\mathtt{z}}]([\mathtt{x} \mapsto \top, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto 0])$.

5.2 Transfer Functions for Conditions

Specification. The interpretation of a condition φ with respect to a given abstract value l must "pass through" all structures that are both represented by l and satisfy φ, i.e., those in $\gamma(l) \cap [\![\varphi]\!]$. Thus, the most-precise approximation to the interpretation of condition φ, denoted by $\mathrm{Assume}^\sharp[\varphi](l)$, is defined by

$$\mathrm{Assume}^\sharp[\varphi](l) = \alpha(\gamma(l) \cap [\![\varphi]\!]).$$

Implementation. $\mathrm{Assume}^\sharp[\varphi](l)$ can be computed by the following method:

$$\mathrm{Assume}^\sharp[\varphi](l) = \widehat{\alpha}(\widehat{\gamma}(l) \wedge \varphi).$$

Example 8.

$$
\begin{array}{rl}
\mathrm{Assume}^\sharp[(y < z)]([\mathtt{x} \mapsto 0, \mathtt{y} \mapsto 2, \mathtt{z} \mapsto 7]) & = \widehat{\alpha}((x = 0) \wedge (y = 2) \wedge (z = 7) \wedge (y < z)) \\
& = [\mathtt{x} \mapsto 0, \mathtt{y} \mapsto 2, \mathtt{z} \mapsto 7] \\
\mathrm{Assume}^\sharp[(y \geq z)]([\mathtt{x} \mapsto 0, \mathtt{y} \mapsto 2, \mathtt{z} \mapsto 7]) & = \widehat{\alpha}((x = 0) \wedge (y = 2) \wedge (z = 7) \wedge (y \geq z)) \\
& = \bot \\
\mathrm{Assume}^\sharp[(y < z)]([\mathtt{x} \mapsto 0, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto 7]) & = \widehat{\alpha}((x = 0) \wedge (z = 7) \wedge (y < z)) \\
& = [\mathtt{x} \mapsto 0, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto 7] \\
\mathrm{Assume}^\sharp[(y = z)]([\mathtt{x} \mapsto 0, \mathtt{y} \mapsto \top, \mathtt{z} \mapsto 7]) & = \widehat{\alpha}((x = 0) \wedge (z = 7) \wedge (y = z)) \\
& = [\mathtt{x} \mapsto 0, \mathtt{y} \mapsto 7, \mathtt{z} \mapsto 7]
\end{array}
$$

6 Discussion

This paper shows how the most-precise versions of the basic operations needed to create an abstract interpreter are, under certain conditions, implementable. These techniques use the idea of considering a first-order formula φ as a device for describing (or accepting) a set of concrete structures, namely, the set of structures that satisfy φ. Not every subset of concrete structures can be described by a first-order formula; however, it is straightforward to generalize the approach to other types of logics, which can be considered as alternative structure-description formalisms (possibly more powerful, possibly less powerful). For the basic approach to carry over, all that is required is that a decision procedure exist for the logic.

Automatic theorem provers—such as MACE [16], SEM [20], and Finder [19]—can be used to implement the procedures presented in this paper because they return counterexamples to validity: a counterexample to the validity of $\neg\varphi$ is a structure that satisfies φ. Such tools also exist for logics other than first-order logic; for example, MONA [15] can generate counterexamples for formulas in weak monadic second-order logic.

Some tools, such as Simplify [9] and SVC [1], provide counterexamples in symbolic form, i.e., as a formula. The formula represents a *set* of counterexamples; any structure that satisfies the formula is a counterexample to the query. For example, if φ is $x \geq y$ at line [5] of Fig. 1, the value returned would be the formula $(x \geq y)$ itself, rather than a particular satisfying structure, such as $[x \mapsto 7, y \mapsto 3]$. This presents an obstacle because at line [6] β requires an argument that is a single structure. In the case of quantifier-free first-order logic with linear arithmetic, such a structure can be obtained by feeding the counterexample formula to a solver for mixed-integer programming, such as CPLEX [13].

```
int x, y, z
Bool B1, B2
y := 3
x := 4 * y + 1
read(z)
B1 := z < 29
B2 := z < 27
if B1 then y := 5
if B2 then x := y + 8
```

Fig. 5. A program with correlated branches.

With the aid of Simplify, we have verified the constant-propagation examples in this paper, as well as examples that combine the constant-propagation domain with a predicate-abstraction domain. This is an additional benefit of the approach: it can be used to generate the best transformer for combined domains, such as reduced cardinal product and those created using other domain constructors [7]. For example, the best transformer for the combined constant-propagation/predicate-abstraction domain determines that the variable x must be 13 at the end of the program given in Fig. 5.

7 Related Work

This paper is most closely related to past work on predicate abstraction, which also uses decision procedures to implement most-precise versions of the basic abstract-interpretation operations. Predicate abstraction only applies to a family of finite-height abstract domains that are finite Cartesian products of Boolean values; our results generalize these ideas to a broader setting. In particular, our work shows that when a small number of conditions are met, most of the benefits that predicate-abstraction domains enjoy can also be enjoyed in arbitrary abstract domains of finite height, and possibly

infinite cardinality. However, procedure $\widehat{\alpha}$ of Fig. 1 uses an approach that is fundamentally different from the one used in predicate abstraction. Although both approaches use multiple calls on a decision procedure to pass from the space of formulas to the domain of abstract values, $\widehat{\alpha}_{\mathrm{PA}}$ goes *directly* from a formula to an abstract value, whereas $\widehat{\alpha}$ of Fig. 1 makes use of the domain of *concrete* values in a critical way: each time around the loop, $\widehat{\alpha}$ selects a concrete value S such that $S \models \varphi$; $\widehat{\alpha}$ uses β and $\sqcup$ to generalize from concrete value S to an abstract value.

Procedure $\widehat{\alpha}$ is also related to an algorithm used in machine learning, called Find-S [17, Section 2.4]. In machine-learning terminology, both algorithms search a space of "hypotheses" to find the most-specific hypothesis that is consistent with the positive training examples of the "concept" to be learned. Find-S receives a sequence of training examples, and generalizes its current hypothesis each time it is presented with a positive training example that falls outside its current hypothesis. The problem settings for the two algorithms are slightly different: Find-S receives a sequence of positive and negative examples of the concept. $\widehat{\alpha}$ already starts with a precise statement of the concept in hand, namely, the formula ψ; on each iteration, the decision procedure is used to generate the next (positive) training example.

We have sometimes been asked "How do your techniques compare with predicate abstraction augmented with an iterative-refinement scheme that generates new predicates, as in SLAM [3] or BLAST [12]?". We do not have a complete answer to this question; however, a few observations can be made:

- Our results extend ideas employed in the setting of predicate abstraction to a more general setting.
- For the simple examples used for illustrative purposes in this paper, iterative refinement would obtain suitable predicates with appropriate constant values in one iteration. Our techniques achieve the desired precision using roughly the same logical machinery (i.e., the availability of a decision procedure), but do not rely on heuristics-based machinery for changing the abstract domain in use.
- This paper studies the problem "How can one obtain most-precise results for a *given* abstract domain?". Iterative refinement addresses a different problem: "How can one go about *improving* an abstract domain?" These are orthogonal questions.

  The question of how to go about improving an abstract domain has not yet been studied for abstract domains as rich as the ones in which our techniques can be applied. This is the subject of future work, and thus something about which one can only speculate. However, we have observed that our approach does provide a fundamental primitive for mapping values from one abstract domain to another: suppose that L_1 and L_2 are two different abstract domains that meet the conditions of the framework; given $l_1 \in L_1$, the most-precise value $l_2 \in L_2$ that overapproximates $\gamma_1(l_1)$ is obtained by $l_2 = \widehat{\alpha}_2(\widehat{\gamma}_1(l_1))$.

  The domain-changing primitive opens up several possibilities for future work. For example, counterexample-guided abstraction-refinement strategies [5,4] identify the shortest invalid prefix of a spurious counterexample trace, and then refine the abstract domain to eliminate invalid transitions out of the last valid abstract state of the prefix. The domain-changing primitive appears to provide a systematic way to salvage information from the counterexample trace: for instance, it can be invoked to convert the last valid abstract state of the prefix into an appropriate abstract state in the refined abstract domain. Moreover, it yields the most-precise value that any conservative salvaging operation is allowed to produce.

In summary, because our results enable a better separation of concerns between the issue of how to obtain most-precise results for a *given* abstract domain and that of how to *improve* an abstract domain, they contribute to a better understanding of abstraction and symbolic approaches to abstract interpretation.

References

1. Stanford validity checker. "http://verify.stanford.edu/SVC/", 1999.
2. T. Ball, R. Majumdar, T. Millstein, and S.K. Rajamani. Automatic predicate abstraction of C programs. In *Prog. Lang. Design and Impl.*, New York, NY, 2001. ACM Press.
3. T. Ball and S.K. Rajamani. The SLAM toolkit. In *Computer-Aided Verif.*, Lec. Notes in Comp. Sci., pages 260–264, 2001.
4. E. Clarke, A. Gupta, J. Kukula, and O. Strichman. SAT based abstraction-refinement using ILP and machine learning techniques. In *Computer-Aided Verif.*, 2002.
5. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer-Aided Verif.*, pages 154–169, July 2000.
6. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Princ. of Prog. Lang.*, 1977.
7. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Princ. of Prog. Lang.*, pages 269–282, New York, NY, 1979. ACM Press.
8. S. Das, D.L. Dill, and S. Park. Experience with predicate abstraction. In *Computer-Aided Verif.*, pages 160–171. Springer-Verlag, July 1999.
9. D. Detlefs, G. Nelson, and J. Saxe. Simplify. Compaq Systems Research Center, Palo Alto, CA, 1999.
10. H.B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.
11. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Computer-Aided Verif.*, pages 72–83, June 1997.
12. T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Princ. of Prog. Lang.*, pages 58–70, New York, NY, January 2002. ACM Press.
13. ILOG. ILOG optimization suite: White paper. ILOG S.A., Gentilly, France, 2001.
14. G.A. Kildall. A unified approach to global program optimization. In *Princ. of Prog. Lang.*, pages 194–206, New York, NY, 1973. ACM Press.
15. N. Klarlund and A. Møller. *MONA Version 1.4 User Manual*. BRICS Notes Series NS-01-1, Univ. of Aarhus, January 2001.
16. W. McCune. *MACE User Manual and Guide*. Argonne Nat. Lab., May 2001.
17. T.M. Mitchell. *Machine Learning*. WCB/McGraw-Hill, Boston, MA, 1997.
18. F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
19. J. Slaney. *Finder – Finite Domain Enumerator, Version 3.0*. Aust. Nat. Univ., July 1995.
20. J. Zhang and H. Zhang. Generating models by SEM. In *Int. Conf. on Auto. Deduc.*, volume 1104 of *Lec. Notes in Art. Intell.*, pages 308–312. Springer-Verlag, 1996.

Constructing Quantified Invariants via Predicate Abstraction*

Shuvendu K. Lahiri and Randal E. Bryant

Carnegie Mellon University, Pittsburgh, PA
shuvendu@ece.cmu.edu,Randy.Bryant@cs.cmu.edu

Abstract. Predicate abstraction provides a powerful tool for verifying properties of infinite-state systems using a combination of a decision procedure for a subset of first-order logic and symbolic methods originally developed for finite-state model checking. We consider models where the system state contains mutable function and predicate state variables. Such a model can describe systems containing arbitrarily large memories, buffers, and arrays of identical processes. We describe a form of predicate abstraction that constructs a formula over a set of universally quantified variables to describe invariant properties of the function state variables. We provide a formal justification of the soundness of our approach and describe how it has been used to verify several hardware and software designs, including a directory-based cache coherence protocol with unbounded FIFO channels.

1 Introduction

Graf and Saïdi introduced *predicate abstraction* [10] as a means of automatically determining invariant properties of infinite-state systems. With this approach, the user provides a set of k Boolean formulas describing possible properties of the system state. These predicates are used to generate a finite state abstraction (containing at most 2^k states) of the system. By performing a reachability analysis of this finite-state model, a predicate abstraction tool can generate the strongest possible invariant for the system expressible in terms of this set of predicates. Prior implementations of predicate abstraction [10,7,1,8] required making a large number of calls to a theorem prover or first-order decision procedure and hence could only be applied to cases where the number of predicates was small. More recently, we have shown that both BDD and SAT-based Boolean methods can be applied to perform the analysis efficiently [12].

In most formulations of predicate abstraction, the predicates contain no free variables; they evaluate to true or false for each system state. The abstraction function α has a simple form, mapping each *concrete* system state to a single *abstract* state based on the effect of evaluating the k predicates. The task of predicate abstraction is to construct a formula ψ^* consisting of some Boolean combination of the predicates such that $\psi^*(s)$ holds for every reachable system state s.

* This research was supported in part by the Semiconductor Research Corporation, Contract RID 1029.

B. Steffen and G. Levi (Eds.): VMCAI 2004, LNCS 2937, pp. 267–281, 2004.

To verify systems containing unbounded resources, such as buffers and memories of arbitrary size and systems with arbitrary number of identical, concurrent processes, the system model must support state variables that are functions or predicates [11,4]. For example, a memory can be represented as a function mapping an address to the data stored at an address, while a buffer can be represented as a function mapping an integer index to the value stored at the specified buffer position. The state elements of a set of identical processes can be modeled as functions mapping an integer process identifier to the state element for the specified process. In many systems, this capability is restricted to *arrays* that can be altered only by writing to a single location [7,11]. Our verifier allows a more general form of mutable function, where the updating operation is expressed using lambda notation.

In verifying systems with function state variables, we require quantified predicates to describe global properties of state variables, such as "At most one process is in its critical section," as expressed by the formula $\forall i, j : \texttt{crit}(i) \wedge \texttt{crit}(j) \Rightarrow i = j$. Conventional predicate abstraction restricts the scope of a quantifier to within an individual predicate. System invariants often involve complex formulas with widely scoped quantifiers. The scoping restriction implies that these invariants cannot be divided into small, simple predicates. This puts a heavy burden on the user to supply predicates that encode intricate sets of properties about the system. Recent work attempts to discover quantified predicates automatically [7], but it has not been successful for many of the systems that we consider.

In this paper we present an extension of predicate abstraction in which the user supplies predicates that include free variables from a set of *index* variables $\mathcal{X}$. The predicate abstraction engine constructs a formula ψ^* consisting of a Boolean combination of these predicates, such that the formula $\forall \mathcal{X} \psi^*(s)$ holds for every reachable system state s. With this method, the predicates can be very simple, with the predicate abstraction tool constructing complex, quantified invariant formulas. For example, the property that at most one process can be in its critical section could be derived by supplying predicates $\texttt{crit(i)}$, $\texttt{crit(j)}$, and $\texttt{i} = \texttt{j}$, where $\texttt{i}$ and $\texttt{j}$ are the index symbols. Encoding these predicates in the abstract system with Boolean variables $\texttt{ci}$, $\texttt{cj}$, and $\texttt{eij}$, respectively, we can verify this property by using predicate abstraction to prove that $\texttt{ci} \wedge \texttt{cj} \Rightarrow \texttt{eij}$ holds for every reachable state of the abstract system.

Flanagan and Qadeer use a method similar to ours [8] for constructing universally quantified loop invariants for sequential software, and we briefly described our method in an earlier paper [12]. Our contribution in this paper is to describe the method more carefully, explore its properties, and to provide a formal argument for its soundness. The key idea of our approach is to formulate the abstraction function α to map a concrete system state s to the set of all possible valuations of the predicates, considering the set of possible values for the index variables $\mathcal{X}$. The resulting abstract system is unusual; it is not characterized by a state transition relation and hence cannot be viewed as a state transition system. Nonetheless, it provides an abstract interpretation of the concrete system [6] that can be used to find invariant system properties.

Assuming a decision procedure that can determine the satisfiability of a formula with universal quantifiers, we can prove the following completeness result for our formulation:

Predicate abstraction can prove any property that can be proved by induction on the state sequence using an induction hypothesis expressed as a universally quantified formula over the given set of predicates. For many modeling logics, this decision problem is undecidable. By using quantifier instantiation, we can implement a sound, but incomplete verifier.

As an extension, we show that it is easy to incorporate *axioms* into the system, properties that must hold universally for every system state. Axioms can be viewed simply as quantified predicates that must evaluate to true on every step. For brevity, this paper only sketches the main proofs. We conclude the paper by describing our use of predicate abstraction to verify several hardware and software systems, including a directory-based cache coherence protocol devised by Steven German [9]. We believe we are the first to verify the protocol for a system with an unbounded number of clients, each communicating via unbounded FIFO channels.

2 Preliminaries

We assume the concrete system is defined in terms of some decidable subset of first-order logic. Our implementation is based on the CLU logic [4], supporting expressions containing uninterpreted functions and predicates, equality and ordering tests, and addition by integer constants, but the ideas of this paper do not depend on the specific modeling formalism. For discussion, we assume that the logic supports Booleans, integers, functions mapping integers to integers, and predicates mapping integers to Booleans.

2.1 Notation

Rather than using the common *indexed vector* notation to represent collections of values (e.g., $\boldsymbol{v} \doteq \langle v_1, v_2, \ldots, v_n \rangle$), we use a *named set* notation. That is, for a set of symbols $\mathcal{A}$, we let $\mathbf{v}$ indicate a set consisting of a value $v_{\mathtt{x}}$ for each $\mathtt{x} \in \mathcal{A}$.

For a set of symbols $\mathcal{A}$, let $\sigma_{\mathcal{A}}$ denote an *interpretation* of these symbols, assigning to each symbol $\mathtt{x} \in \mathcal{A}$ a value $\sigma_{\mathcal{A}}(\mathtt{x})$ of the appropriate type (Boolean, integer, function, or predicate). Let $\Sigma_{\mathcal{A}}$ denote the set of all interpretations $\sigma_{\mathcal{A}}$ over the symbol set $\mathcal{A}$.

For interpretations $\sigma_{\mathcal{A}}$ and $\sigma_{\mathcal{B}}$ over disjoint symbol sets $\mathcal{A}$ and $\mathcal{B}$, let $\sigma_{\mathcal{A}} \cdot \sigma_{\mathcal{B}}$ denote an interpretation assigning either $\sigma_{\mathcal{A}}(\mathtt{x})$ or $\sigma_{\mathcal{B}}(\mathtt{x})$ to each symbol $\mathtt{x} \in \mathcal{A} \cup \mathcal{B}$, according to whether $\mathtt{x} \in \mathcal{A}$ or $\mathtt{x} \in \mathcal{B}$.

For symbol set $\mathcal{A}$, let $E(\mathcal{A})$ denote the set of all expressions in the logic over $\mathcal{A}$. For any expression $e \in E(\mathcal{A})$ and interpretation $\sigma_{\mathcal{A}} \in \Sigma_{\mathcal{A}}$, let the *valuation of* e *with respect to* $\sigma_{\mathcal{A}}$, denoted $\langle e \rangle_{\sigma_{\mathcal{A}}}$ be the (Boolean, integer, function, or predicate) value obtained by evaluating e when each symbol $\mathtt{x} \in \mathcal{A}$ is replaced by its interpretation $\sigma_{\mathcal{A}}(\mathtt{x})$.

Let $\mathbf{v}$ be a named set over symbols $\mathcal{A}$, consisting of expressions over symbol set $\mathcal{B}$. That is, $v_{\mathtt{x}} \in E(\mathcal{B})$ for each $\mathtt{x} \in \mathcal{A}$. Given an interpretation $\sigma_{\mathcal{B}}$ of the symbols in $\mathcal{B}$, evaluating the expressions in $\mathbf{v}$ defines an interpretation of the symbols in $\mathcal{A}$, which we

denote $\langle \mathbf{v} \rangle_{\sigma_\mathcal{B}}$. That is, $\langle \mathbf{v} \rangle_{\sigma_\mathcal{B}}$ is an interpretation $\sigma_\mathcal{A}$ such that $\sigma_\mathcal{A}(\mathtt{x}) = \langle v_\mathtt{x} \rangle_{\sigma_\mathcal{B}}$ for each $\mathtt{x} \in \mathcal{A}$.

A *substitution* $\boldsymbol{\pi}$ for a set of symbols $\mathcal{A}$ is a named set of expressions over some set of symbols $\mathcal{B}$ (with no restriction on the relation between $\mathcal{A}$ and $\mathcal{B}$.) That is, for each $\mathtt{x} \in \mathcal{A}$, there is an expression $\pi_\mathtt{x} \in E(\mathcal{B})$. For an expression $e \in E(\mathcal{A} \cup \mathcal{C})$, we let $e\,[\boldsymbol{\pi}/\mathcal{A}]$ denote the expression $e' \in E(\mathcal{B} \cup \mathcal{C})$ resulting when we (simultaneously) replace each occurrence of every symbol $\mathtt{x} \in \mathcal{A}$ with the expression $\pi_\mathtt{x}$.

2.2 System Model

We model the system as having a number of *state elements*, where each state element may be a Boolean or integer value, or a function or predicate. We use symbolic names to represent the different state elements giving the set of *state symbols* $\mathcal{V}$. We introduce a set of *initial state* symbols $\mathcal{J}$ and a set of *input* symbols $\mathcal{I}$ representing, respectively, initial values and inputs that can be set to arbitrary values on each step of operation. Among the state variables, there can be *immutable* values expressing the behavior of functional units, such as ALUs, and system parameters such as the total number of processes or the maximum size of a buffer. Since these values are expressed symbolically, one run of the verifier can prove the correctness of the system for arbitrary functionalities, process counts, and buffer capacities.

The overall system operation is characterized by an *initial-state* expression set $\mathbf{q}^0$ and a *next-state* expression set $\boldsymbol{\delta}$. The initial state consists of an expression for each state element, with the initial value of state element $\mathtt{x}$ given by expression $q^0_\mathtt{x} \in E(\mathcal{J})$. The transition behavior also consists of an expression for each state element, with the behavior for state element $\mathtt{x}$ given by expression $\delta_\mathtt{x} \in E(\mathcal{V} \cup \mathcal{I})$. In this expression, the state element symbols represent the current system state and the input symbols represent the current values of the inputs. The expression gives the new value for that state element.

We will use a very simple system as a running example throughout this presentation. The only state element is a function $\mathtt{F}$. An input symbol $\mathtt{i}$ determines which element of $\mathtt{F}$ is updated. Initially, $\mathtt{F}$ is the identify function: $q^0_\mathtt{F} = \lambda\, u\, .\, u$. On each step, the value of the function for argument $\mathtt{i}$ is updated to be $\mathtt{F}(\mathtt{i}{+}1)$. That is, $\delta_\mathtt{F} = \lambda\, u\, .\, \mathit{ITE}(u = \mathtt{i},\, \mathtt{F}(\mathtt{i}{+}1),\, \mathtt{F}(u))$, where the if-then-else operation *ITE* selects its second argument when the first one evaluates to true and the third otherwise.

2.3 Concrete System

A concrete system state assigns an interpretation to every state symbol. The set of states of the concrete system is given by $\Sigma_\mathcal{V}$, the set of interpretations of the state element symbols. For convenience, we denote concrete states using letters s and t rather than the more formal $\sigma_\mathcal{V}$.

From our system model, we can characterize the behavior of the concrete system in terms of an initial state set $Q^0_C \subseteq \Sigma_\mathcal{V}$ and a next-state function operating on sets $N_C \colon \mathrm{P}(\Sigma_\mathcal{V}) \to \mathrm{P}(\Sigma_\mathcal{V})$. The initial state set is defined as $Q^0_C \doteq \{\langle \mathbf{q}^0 \rangle_{\sigma_\mathcal{J}} \,|\sigma_\mathcal{J} \in \Sigma_\mathcal{J}\}$,

i.e., the set of all possible valuations of the initial state expressions. The next-state function N_C is defined for a single state s as $N_C(s) \doteq \{\langle \boldsymbol{\delta} \rangle_{s \cdot \sigma_{\mathcal{I}}} | \sigma_{\mathcal{I}} \in \Sigma_{\mathcal{I}}\}$, i.e., the set of all valuations of the next-state expressions for concrete state s and arbitrary input. The function is then extended to sets of states by defining $N_C(S_C) = \bigcup_{s \in S_C} N_C(s)$. We can also characterize the next-state behavior of the concrete system by a transition relation T where $(s,t) \in T$ when $t \in N_C(s)$.

We define the set of reachable states R_C as containing those states s such that there is some state sequence $s_0, s_1, \ldots, s_n$ with $s_0 \in Q_C^0$, $s_n = s$, and $s_{i+1} \in N_C(s_i)$ for all values of i such that $0 \leq i < n$. We define the *depth* of a reachable state s to be the length n of the shortest sequence leading to s. Since our concrete system has an infinite number of states, there is no finite bound on the maximum depth over all reachable states.

With our example system, the concrete state set consists of integer functions f such that $f(u+1) \geq f(u) \geq u$ for all u and $f(u) = u$ for infinitely many arguments of f.

3 Predicate Abstraction

We use quantified predicates to express constraints on the system state. To define the abstract state space, we introduce a set of *predicate* symbols $\mathcal{P}$ and a set of *index* symbols $\mathcal{X}$. The predicates consist of a named set $\boldsymbol{\phi}$, where for each $\mathtt{p} \in \mathcal{P}$, predicate $\phi_{\mathtt{p}}$ is a Boolean formula over the symbols in $\mathcal{V} \cup \mathcal{X}$.

Our predicates define an abstract state space $\Sigma_{\mathcal{P}}$, consisting of all interpretations $\sigma_{\mathcal{P}}$ of the predicate symbols. For $k \doteq |\mathcal{P}|$, the state space contains 2^k elements.

As an illustration, suppose for our example system we wish to prove that state element $\mathtt{F}$ will always be a function f satisfying $f(u) \geq 0$ for all $u \geq 0$. We introduce an index variable $\mathtt{x}$ and predicate symbols $\mathcal{P} = \{\mathtt{p}, \mathtt{q}\}$, with $\phi_{\mathtt{p}} \doteq \mathtt{F}(\mathtt{x}) \geq 0$ and $\phi_{\mathtt{q}} \doteq \mathtt{x} \geq 0$.

We can denote a set of abstract states by a Boolean formula $\psi \in E(\mathcal{P})$. This expression defines a set of states $\langle \psi \rangle \doteq \{\sigma_{\mathcal{P}} | \langle \psi \rangle_{\sigma_{\mathcal{P}}} = \mathbf{true}\}$. As an example, our two predicates $\phi_{\mathtt{p}}$ and $\phi_{\mathtt{q}}$ generate an abstract space consisting of four elements, which we denote FF, FT, TF, and TT, according to the interpretations assigned to $\mathtt{p}$ and $\mathtt{q}$. There are then 16 possible abstract state sets, some of which are shown in Table 1. In this table, abstract state sets are represented both by Boolean formulas over $\mathtt{p}$ and $\mathtt{q}$, and by enumerations of the state elements.

We define the *abstraction function* α to map each concrete state to the set of abstract states given by the valuations of the predicates for all possible values of the index variables:

$$\alpha(s) \doteq \left\{ \langle \boldsymbol{\phi} \rangle_{s \cdot \sigma_{\mathcal{X}}} | \sigma_{\mathcal{X}} \in \Sigma_{\mathcal{X}} \right\} \tag{1}$$

Since there are multiple interpretations $\sigma_{\mathcal{X}}$, a single concrete state will generally map to multiple abstract states. This feature is not found in most uses of predicate abstraction, but it is the key idea for handling quantified predicates. We then extend the abstraction function to apply to sets of concrete states in the usual way: $\alpha(S_C) \doteq \bigcup_{s \in S_C} \alpha(s)$. We can see that α is monotonic, i.e., that if $S_C \subseteq T_C$, then $\alpha(S_C) \subseteq \alpha(T_C)$.

Table 1. Example abstract state sets and their concretizations Abstract state elements are represented by their interpretations of p and q. The terms are interpreted over $\mathcal{Z}$.

Abstract System		Concrete System	
Formula ψ	State Set $S_A = \langle\psi\rangle$	System Property $\forall\mathcal{X}\psi^*$	State Set $S_C = \gamma(S_A)$
$\mathtt{p} \wedge \mathtt{q}$	{TT}	$\forall \mathtt{x} : \mathtt{F(x)} \geq 0 \wedge \mathtt{x} \geq 0$	$\emptyset$
$\mathtt{p} \wedge \neg\mathtt{q}$	{TF}	$\forall \mathtt{x} : \mathtt{F(x)} \geq 0 \wedge \mathtt{x} < 0$	$\emptyset$
$\neg\mathtt{q}$	{FF, TF}	$\forall \mathtt{x} : \mathtt{x} < 0$	$\emptyset$
p	{TF, TT}	$\forall \mathtt{x} : \mathtt{F(x)} \geq 0$	$\{f \mid \forall x : f(x) \geq 0\}$
$\mathtt{p} \vee \neg\mathtt{q}$	{FF, TF, TT}	$\forall \mathtt{x} : \mathtt{x} \geq 0 \Rightarrow \mathtt{F(x)} \geq 0$	$\{f \mid \forall x : x \geq 0 \Rightarrow f(x) \geq 0\}$

Working with our example system, consider the concrete state given by the function $\lambda\, u\, .\, u$. When we abstract this function relative to predicates $\phi_{\mathtt{p}}$ and $\phi_{\mathtt{q}}$, we get two abstract states: TT, when $\mathtt{x} \geq 0$, and FF, when $\mathtt{x} < 0$. This abstract state set is then characterized by the formula $\mathtt{p} \Leftrightarrow \mathtt{q}$.

We define the concretization function γ to require universal quantification over the index symbols. That is, for a set of abstract states $S_A \subseteq \Sigma_{\mathcal{P}}$:

$$\gamma(S_A) \quad \doteq \quad \left\{s \mid \forall \sigma_{\mathcal{X}} \in \Sigma_{\mathcal{X}} : \langle \boldsymbol{\phi} \rangle_{s \cdot \sigma_{\mathcal{X}}} \in S_A\right\} \tag{2}$$

The universal quantifier in this definition has the consequence that the concretization function does not distribute over set union. In particular, we cannot view the concretization function as operating on individual abstract states, but rather as generating each concrete state from multiple abstract states. Nonetheless, γ is montonic, i.e., if $S_A \subseteq T_A$, then $\gamma(S_A) \subseteq \gamma(T_A)$.

Consider our example system with predicates $\phi_{\mathtt{p}}$ and $\phi_{\mathtt{q}}$. Table 1 shows some example abstract state sets S_A and their concretizations $\gamma(S_A)$. As the first three examples show, some (altogether 6) nonempty abstract state sets have empty concretizations, because they constrain x to be either always negative or always non-negative. On the other hand, there are 9 abstract state sets having nonempty concretizations. We can see by this that the concretization function is based on the entire abstract state set and not just on the individual values. For example, the sets {TF} and {TT} have empty concretizations, but {TF, TT} concretizes to the set of all non-negative functions.

Theorem 1. *The functions (α, γ) form a Galois connection* [1]*, i.e., for any sets of concrete states S_C and abstract states S_A:*

$$\alpha(S_C) \subseteq S_A \;\Leftrightarrow\; S_C \subseteq \gamma(S_A) \tag{3}$$

The proof follows by observing that both the left and the right-hand sides of (3) hold precisely when for every $\sigma_{\mathcal{X}} \in \Sigma_{\mathcal{X}}$ and every $s \in S_C$, we have $\langle \boldsymbol{\phi} \rangle_{s \cdot \sigma_{\mathcal{X}}} \in S_A$.

[1] This is one of several logically equivalent formulations of a Galois connection [6].

A Galois connection also satisfies the property (follows from (3)) that for any set of concrete states S_C:

$$S_C \subseteq \gamma(\alpha(S_C)). \tag{4}$$

The containment relation in (4) can be proper. For example, the concrete state set consisting of the single function $\lambda u \,.\, u$ abstracts to the state set $\mathtt{p} \Leftrightarrow \mathtt{q}$, which in turn concretizes to the set of all functions f such that $f(u) \geq 0 \Leftrightarrow u \geq 0$.

4 Abstract System

Predicate abstraction involves performing a reachability analysis over the abstract state space, where on each step we concretize the abstract state set via γ, apply the concrete next-state function, and then abstract the results via α. We can view this process as performing reachability analysis on an abstract system having initial state set $Q_A^0 \doteq \alpha(Q_C^0)$ and a next-state function operating on sets: $N_A(S_A) \doteq \alpha(N_C(\gamma(S_A)))$. Note that there is no transition relation associated with this next-state function, since γ cannot be viewed as operating on individual abstract states.

It can be seen that N_A provides an *abstract interpretation* of the concrete system [6,5]:

1. N_A is null-preserving: $N_A(\emptyset) = \emptyset$
2. N_A is monotonic: $S_A \subseteq T_A \;\Rightarrow\; N_A(S_A) \subseteq N_A(T_A)$
3. N_A simulates N_C (a simulation relation defined by α): $\alpha(N_C(S_C)) \subseteq N_A(\alpha(S_C))$

We perform reachability analysis on the abstract system using N_A as the next-state function:

$$R_A^0 = Q_A^0 \tag{5}$$
$$R_A^{i+1} = R_A^i \cup N_A(R_A^i) \tag{6}$$

Since the abstract system is finite, there must be some n such that $R_A^n = R_A^{n+1}$. The set of all reachable abstract states R_A is then R_A^n. By induction on n, it can be shown that if s is a reachable state in the concrete system with depth $\leq n$, then $\alpha(s) \subseteq R_A^n$. From this it follows that $\alpha(s) \subseteq R_A$ for any concrete reachable state s, and therefore that $\alpha(R_C) \subseteq R_A$. Thus, even though determining the set of reachable concrete states would require examining paths of unbounded length, we can compute a conservative approximation to this set by performing a bounded reachability analysis of the abstract system.

It is worth noting that we cannot use the standard "frontier set" optimization in our reachability analysis. This optimization, commonly used in symbolic model checking, considers only the newly reached states in computing the next set of reachable states. In our context, this would mean using the computation $R_A^{i+1} = R_A^i \cup N_A(R_A^i - R_A^{i-1})$ rather than that of (6). This optimization is not valid, due to the fact that γ, and therefore N_A, does not distribute over set union.

As an illustration, let us perform reachability analysis on our example system. In the initial state, state element F is the identity function, which we have seen abstracts to the set represented by the formula $\mathtt{p} \Leftrightarrow \mathtt{q}$. This abstract state set concretizes to the set of functions f satisfying $f(u) \geq 0 \Leftrightarrow u \geq 0$. Let h denote the value of F in the next state. If input i is -1, we would $h(-1) = f(0) \geq 0$, but we can still guarantee that $h(u) \geq 0$ for $u \geq 0$. Applying the abstraction function, we get R_A^1 characterized by the formula $\mathtt{p} \vee \neg\mathtt{q}$ (see Table 1.) For the second iteration, the abstract state set characterized by the formula $\mathtt{p} \vee \neg\mathtt{q}$ concretizes to the set of functions f satisfying $f(u) \geq 0$ when $u \geq 0$, and this condition must hold in the next state as well. Applying the abstraction function to this set, we then get $R_A^2 = R_A^1$, and hence the process has converged.

5 Verifying Safety Properties

A Boolean formula $\psi \in E(\mathcal{P})$ defines a *property* of the abstract state space. The property is said to hold for the abstract system when it holds for every reachable abstract state. That is, $\langle\psi\rangle_{\sigma_\mathcal{P}} = \mathbf{true}$ for all $\sigma_\mathcal{P} \in R_A$.

For Boolean formula $\psi \in E(\mathcal{P})$, define the formula $\psi^* \in E(\mathcal{V} \cup \mathcal{X})$ to be the result of substituting the predicate expression $\phi_\mathtt{p}$ for each predicate symbol $\mathtt{p} \in \mathcal{P}$. That is, viewing $\boldsymbol{\phi}$ as a substitution, we have $\psi^* \doteq \psi\,[\boldsymbol{\phi}/\mathcal{P}]$. Formula ψ^* defines a property $\forall\mathcal{X}\psi^*$ of the concrete states. The property holds for concrete state s, written $\forall\mathcal{X}\psi^*(s)$, when $\langle\psi^*\rangle_{s\cdot\sigma_\mathcal{X}} = \mathbf{true}$ for every $\sigma_\mathcal{X} \in \Sigma_\mathcal{X}$. The property holds for the concrete system when $\forall\mathcal{X}\psi^*(s)$ holds for every reachable concrete state $s \in R_C$. Table 1 shows the concrete system properties given by different abstract state formulas ψ.

Theorem 2. *For a formula $\psi \in E(\mathcal{P})$, if property ψ holds for the abstract system, then property $\forall\mathcal{X}\psi^*$ holds for the concrete system.*

This follows by the definition of α and the fact that $\alpha(R_C) \subseteq R_A$.

With our example system, letting formula $\psi \doteq \mathtt{p} \vee \neg\mathtt{q}$, and noting that $\mathtt{p} \vee \neg\mathtt{q} \equiv \mathtt{q} \Rightarrow \mathtt{p}$, we get the property $\forall \mathtt{x} : \mathtt{x} \geq 0 \Rightarrow \mathtt{F}(\mathtt{x}) \geq 0$.

Using predicate abstraction, we can possibly get a *false negative* result, where we fail to verify a property $\forall\mathcal{X}\psi^*$, even though it holds for the concrete system, because the given set of predicates does not adequately capture the characteristics of the system that ensure the desired property. Thus, this method of verifying properties is sound, but possibly incomplete.

We can precisely characterize the class of properties for which the predicate abstraction is both sound and complete, assuming we have a decision procedure that can determine whether a universally quantified formula in the underlying logic is satisfiable. A property $\forall\mathcal{X}\psi^*$ is said to be *inductive* for the concrete system when it satisfies the following two properties:

1. Every initial state $s \in Q_C^0$ satisfies $\forall\mathcal{X}\psi^*(s)$.
2. For all concrete states s and $t \in N_C(s)$, if $\forall\mathcal{X}\psi^*(s)$ holds, then so does $\forall\mathcal{X}\psi^*(t)$.

Clearly an inductive property must hold for every reachable concrete state and therefore for the concrete system. It can also be shown that if $\forall\mathcal{X}\psi^*$ is inductive, then ψ holds for the abstract system. That is, if we present the predicate abstraction engine with a fully formed induction hypothesis, it can prove that it holds.

For formula $\psi \in E(\mathcal{P})$ and predicate set $\boldsymbol{\phi}$, the property $\forall\mathcal{X}\psi^*$ is said to *have an induction proof over* $\boldsymbol{\phi}$ when there is some formula $\chi \in E(\mathcal{P})$, such that $\chi \Rightarrow \psi$ and $\forall\mathcal{X}\chi^*$ is inductive. That is, there is some way to strengthen ψ into a formula χ that can be used to prove the property by induction.

Theorem 3. *A formula $\psi \in E(\mathcal{P})$ is a property of the abstract system if and only if the concrete property $\forall\mathcal{X}\psi^*$ has an induction proof over the predicate set $\boldsymbol{\phi}$.*

This theorem precisely characterizes the capability of our formulation of predicate abstraction—it can prove any property that can be proved by induction using an induction hypothesis expressed in terms of the predicates. Thus, if we fail to verify a system using this form of predicate abstraction, we can conclude that either 1) the system does not satisfy the property, or 2) we did not provide an adequate set of predicates to construct an universally quantified induction hypothesis, provided one exists.

6 Quantifier Instantiation

For many subsets of first-order logic, there is no complete method for handling the universal quantifier introduced in function γ (Equation 2). For example, in a logic with uninterpreted functions and equality, determining whether a universally quantified formula is satisfiable is undecidable [3]. Instead, we concretize abstract states by considering some limited subset of the interpretations of the index symbols, each of which is defined by a substitution for the symbols in $\mathcal{X}$. Our tool automatically generates candidate substitutions based on the subexpressions that appear in the predicate and next-state expressions [13]. These subexpressions can contain symbols in $\mathcal{V}$, $\mathcal{X}$, and $\mathcal{I}$. These instantiated versions of the formulas enable to verifier to detect specific cases where the predicates can be applied. Flanagan and Qadeer use a similar technique [8].

More precisely, let $\boldsymbol{\pi}$ be a substitution assigning an expression $\pi_{\mathtt{x}} \in E(\mathcal{V} \cup \mathcal{X} \cup \mathcal{I})$ for each $\mathtt{x} \in \mathcal{X}$. Then $\phi_{\mathtt{p}}\,[\boldsymbol{\pi}/\mathcal{X}]$ will be a Boolean expression over symbols $\mathcal{V}$, $\mathcal{X}$, and $\mathcal{I}$ that represents some instantiation of predicate $\phi_{\mathtt{p}}$. For a set of substitutions Π and interpretations $\sigma_{\mathcal{X}} \in \Sigma_{\mathcal{X}}$ and $\sigma_{\mathcal{I}} \in \Sigma_{\mathcal{I}}$, we define the concretization function γ_{Π} as:

$$\gamma_{\Pi}(S_A, \sigma_{\mathcal{X}}, \sigma_{\mathcal{I}}) \quad \doteq \quad \left\{s | \forall\boldsymbol{\pi} \in \Pi : \langle\boldsymbol{\phi}\,[\boldsymbol{\pi}/\mathcal{X}]\rangle_{s\cdot\sigma_{\mathcal{X}}\cdot\sigma_{\mathcal{I}}} \in S_A\right\} \tag{7}$$

It can be seen that γ_{Π} is an overapproximation of γ, i.e., that $\gamma(S_A) \subseteq \gamma_{\Pi}(S_A, \sigma_{\mathcal{X}}, \sigma_{\mathcal{I}})$ for any abstract state S_A, set of substitutions Π, and interpretations $\sigma_{\mathcal{X}}$ and $\sigma_{\mathcal{I}}$. From (4), it then follows that

$$S_C \quad \subseteq \quad \gamma_{\Pi}(\alpha(S_C), \sigma_{\mathcal{X}}, \sigma_{\mathcal{I}}). \tag{8}$$

and hence the functions (α, γ_{Π}) satisfy property (4) of a Galois connection, even though they are not a true Galois connection.

We can use γ_{Π} as an approximation to γ in defining the behavior of the abstract system. That is, define N_{Π} over sets of abstract states as:

$$N_{\Pi}(S_A) = \left\{ \langle \boldsymbol{\phi}\left[\boldsymbol{\delta}/\mathcal{V}\right] \rangle_{s \cdot \sigma_{\mathcal{X}} \cdot \sigma_{\mathcal{I}}} \,|\sigma_{\mathcal{X}} \in \Sigma_{\mathcal{X}}, \sigma_{\mathcal{I}} \in \Sigma_{\mathcal{I}}, s \in \gamma_{\Pi}(S_A, \sigma_{\mathcal{X}}, \sigma_{\mathcal{I}}) \right\} \quad (9)$$

Observe in this equation that $\phi_{\mathtt{p}}\left[\boldsymbol{\delta}/\mathcal{V}\right]$ is an expression describing the evaluation of predicate $\phi_{\mathtt{p}}$ in the next state. It can be seen that $N_{\Pi}(S_A) \supseteq N_A(S_A)$ for any set of abstract states S_A. As long as Π is nonempty (required to guarantee that N_{Π} is null-preserving), it can be shown that the system defined by N_{Π} is an abstract interpretation of the concrete system. We can therefore perform reachability analysis:

$$R^0_{\Pi} = Q^0_A \quad (10)$$

$$R^{i+1}_{\Pi} = R^i_{\Pi} \cup N_{\Pi}(R^i_{\Pi}) \quad (11)$$

These iterations will converge to a set R_{Π}. For every step i, we can see that $R^i_{\Pi} \supseteq R^i_A$, and therefore we must have $R_{\Pi} \supseteq R_A$.

Theorem 4. *For a formula $\psi \in E(\mathcal{P})$, if $\langle \psi \rangle_{\sigma_{\mathcal{P}}} = \mathbf{true}$ for every $\sigma_{\mathcal{P}} \in R_{\Pi}$, then property $\forall \mathcal{X} \psi^*$ holds for the concrete system.*

This demonstrates that using quantifier instantiation during reachability analysis yields a sound verification technique. However, when the tool fails to verify a property, it could mean, in addition to the two possibilities listed earlier, that 3) it used an inadequate set of instantiations, or 4) that the property cannot be proved by any bounded set of quantifier instantiations.

7 Symbolic Formulation of Reachability Analysis

We are now ready to express the reachability computation symbolically, where each step involves finding the set of satisfying solutions to an existentially quantified formula. On each step, we generate a Boolean formula ρ^i_{Π}, that characterizes R^i_{Π}. That is $\langle \rho^i_{\Pi} \rangle = R^i_{\Pi}$. The formulas directly encode the approximate reachability computations of (10) and (11).

Observe that by composing the predicate expressions with the initial state expressions, $\boldsymbol{\phi}\left[\mathbf{q}^0/\mathcal{V}\right]$, we get a set of predicates over the initial state symbols $\mathcal{J}$ indicating the conditions under which the predicates hold in the initial state. We can therefore start the reachability analysis by finding solutions to the formula

$$\rho^0_{\Pi}(\mathcal{P}) = \exists \mathcal{J} \exists \mathcal{X} \bigwedge_{\mathtt{p} \in \mathcal{P}} \mathtt{p} \Leftrightarrow \phi_{\mathtt{p}}\left[\mathbf{q}^0/\mathcal{V}\right] \quad (12)$$

The formula for the next-state computation combines the definitions of N_Π (9) and γ_Π (7):

$$\rho_\Pi^{i+1}(\mathcal{P}) = \rho_\Pi^i(\mathcal{P}) \vee \exists\mathcal{V}\exists\mathcal{X}\exists\mathcal{I}\left(\bigwedge_{\boldsymbol{\pi}\in\Pi}\left(\rho_\Pi^i\left[\boldsymbol{\phi}/\mathcal{P}\right]\right)\left[\boldsymbol{\pi}/\mathcal{X}\right] \;\wedge\; \bigwedge_{\mathtt{p}\in\mathcal{P}} \mathtt{p} \Leftrightarrow \phi_{\mathtt{p}}\left[\boldsymbol{\delta}/\mathcal{V}\right]\right). \quad (13)$$

To understand the quantified term in this equation, note that the left-hand term is the formula for $\gamma_\Pi(\rho_\Pi^i, \sigma_\mathcal{X}, \sigma_\mathcal{I})$, while the right-hand term expresses the conditions under which each abstract state variable $\mathtt{p}$ will match the value of the corresponding predicate in the next state.

Let us see how this symbolic formulation would perform reachability analysis for our example system. Recall that our system has two predicates $\phi_\mathtt{p} \doteq \mathtt{F(x)} \geq 0$ and $\phi_\mathtt{q} \doteq \mathtt{x} \geq 0$. In the initial state, $\mathtt{F}$ is the function $\lambda\, u\, .\, u$, and therefore $\phi_\mathtt{p}\left[\mathbf{q}^0/\mathcal{V}\right]$ simply becomes $\mathtt{x} \geq 0$. Equation (12) then becomes $\exists \mathtt{x}\,[(\mathtt{p} \Leftrightarrow \mathtt{x} \geq 0) \wedge (\mathtt{q} \Leftrightarrow \mathtt{x} \geq 0)]$, which reduces to $\mathtt{p} \Leftrightarrow \mathtt{q}$.

Now let us perform the first iteration. For our instantiations we require two substitutions $\boldsymbol{\pi}$ and $\boldsymbol{\pi}'$ with $\pi_\mathtt{x} = \mathtt{x}$ and $\pi'_\mathtt{x} = \mathtt{i}{+}1$. For $\rho_\Pi^0(\mathtt{p},\mathtt{q}) = \mathtt{p} \Leftrightarrow \mathtt{q}$, the left-hand term of (13) instantiates to $(\mathtt{F(x)} \geq 0 \Leftrightarrow \mathtt{x} \geq 0) \wedge (\mathtt{F(i{+}1)} \geq 0 \Leftrightarrow \mathtt{i{+}1} \geq 0)$. Substituting $\lambda\, u.\mathit{ITE}(u = \mathtt{i},\, \mathtt{F(i{+}1)},\, \mathtt{F}(u))$ for $\mathtt{F}$ in $\phi_\mathtt{p}$ gives $(\mathtt{x{=}i} \wedge \mathtt{F(i{+}1)} \geq 0) \vee (\mathtt{x{\neq}i} \wedge \mathtt{F(x)} \geq 0)$.

The quantified portion of (13) for $\rho_\Pi^1(\mathtt{p},\mathtt{q})$ then becomes

$$\exists\, \mathtt{F},\mathtt{x},\mathtt{i} : \begin{pmatrix} \mathtt{F(x)} \geq 0 \Leftrightarrow \mathtt{x} \geq 0 \;\wedge\; \mathtt{F(i{+}1)} \geq 0 \Leftrightarrow \mathtt{i{+}1} \geq 0 \\ \wedge\, \mathtt{p} \Leftrightarrow [(\mathtt{x{=}i} \wedge \mathtt{F(i{+}1)} \geq 0) \vee (\mathtt{x{\neq}i} \wedge \mathtt{F(x)} \geq 0)] \\ \wedge\, \mathtt{q} \Leftrightarrow \mathtt{x} \geq 0 \end{pmatrix}$$

The only values of $\mathtt{p}$ and $\mathtt{q}$ where this formula cannot be satisfied is when $\mathtt{p}$ is false and $\mathtt{q}$ is true.

As shown in [12], we can generate the set of solutions to (12) and (13) by first transforming the formulas into equivalent Boolean formulas and then performing Boolean quantifier elimination to remove all Boolean variables other than those in $\mathcal{P}$. This quantifier elimination is similar to the relational product operation used in symbolic model checking and can be solved using either BDD or SAT-based methods.

8 Axioms

As a special class of predicates, we may have some that are to hold at all times. For example, we could have an axiom $\mathtt{f(w)} > 0$ to indicate that function $\mathtt{f}$ is always positive, or $\mathtt{f(y,z)} = \mathtt{f(z,y)}$ to indicate that $\mathtt{f}$ is commutative. Typically, we want these predicates to be individually quantified, but we can ensure this by defining each of them over a unique set of index symbols, as we have done in the above examples.

We can add this feature to our analysis by identifying a subset $\mathcal{Q}$ of the predicate symbols $\mathcal{P}$ to be axioms. We then want to restrict the analysis to states where the axiomatic predicates hold. Let $\Sigma_{\mathcal{P}}^{\mathcal{Q}}$ denote the set of abstract states $\sigma_{\mathcal{P}}$ where $\sigma_{\mathcal{P}}(\mathtt{p}) = \mathbf{true}$ for every $\mathtt{p} \in \mathcal{Q}$. Then we can apply this restriction by redefining $\alpha(s)$ (Equation 1) for concrete state s to be:

$$\alpha(s) \doteq \left\{ \langle \boldsymbol{\phi} \rangle_{s \cdot \sigma_{\mathcal{X}}} \mid \sigma_{\mathcal{X}} \in \Sigma_{\mathcal{X}} \right\} \cap \Sigma_{\mathcal{P}}^{\mathcal{Q}} \tag{14}$$

and then using this definition in the extension of α to sets, the formulation of the reachability analysis (Equations 5 and 6), and the approximate reachability analysis (Equations 10 and 11).

9 Applications

We have used our predicate abstraction tool to verify safety properties of a variety of models and protocols. Some of the more interesting ones include:

- A microprocessor out-of-order execution unit with an unbounded retirement buffer. Prior verification of this unit required manually generating 13 invariants [13].
- A directory-based cache protocol with unbounded channels, devised by Steven German of IBM [9], as discussed below.
- A version of Lamport's bakery algorithm [14] that allows arbitrary number of processes and nonatomic reads and writes.
- Selection sort algorithm for sorting an arbitrary large array. We prove the property that upon termination, the algorithm produces a sorted array.

For the directory-based German's cache-coherence protocol, an unbounded number of clients (`cache`), communicate with a central *home* process to gain *exclusive* or *shared* access to a memory line. The state of each `cache` can be {INVALID, SHARED, EXCLUSIVE}. The home maintains explicit representations of two lists of clients: those sharing the cache line (`home_sharer_list`) and those for which the home has sent an invalidation request but has not received an acknowledgement (`home_invalidate_list`).

The client places requests {REQ_SHARED, REQ_EXCLUSIVE} on a channel `ch_1` and the home grants {GRANT_SHARED, GRANT_EXCLUSIVE} on channel `ch_2`. The home also sends invalidation messages INVALIDATE along `ch_2`. The home grants exclusive access to a client only when there are no clients sharing a line, i.e. $\forall \mathtt{i} : \mathtt{home\_sharer\_list(i)} = \mathbf{false}$. The home maintains variables for the current client (`home_current_client`) and the request it is currently processing (`home_current_command`). It also maintains a bit `home_exclusive_granted` to indicate that some client has exclusive access. The cache lines acknowledge invalidation requests with a INVALIDATE_ACK along another channel `ch_3`. Details of the protocol operation with single-entry channels can be found in many previous works including [15].

In our version of the protocol, each `cache` communicates to the home process through three directed unbounded FIFO channels, namely the channels `ch_1`, `ch_2`, `ch_3`. Thus,

there are an unbounded number of unbounded channels, three for each client[2]. It can be shown that a client can generate an unbounded number of requests before getting a response from the home.

To model the protocol in CLU, we need to change the predicate state variable representation of home_sharer_list. Since the transition functions are expressed over quantifier-free logic, we cannot support a universal quantifier in the model. Instead, we model home_sharer_list as a *set*, using (1) a queue $\texttt{hsl\_q} \doteq \langle \texttt{q}, \texttt{hd}, \texttt{tl} \rangle$ to store all cache indices i for which $\texttt{home\_sharer\_list}(\texttt{i}) = \mathbf{true}$ and (2) an array hsl_pos to map a cache index i to the position in the queue, if $\texttt{i} \in \texttt{hsl\_q}$. This representation can support addition, deletion, membership-check and emptiness-check, which are the operations required for this protocol. In addition, this representation also allows us to *enumerate* the cache indices for which $\texttt{home\_sharer\_list}(\texttt{i}) = \mathbf{true}$.

We had previously verified the cache-coherence property of the protocol with 31 non-trivial, manually constructed invariants. In contrast, the predicate abstraction constructs the strongest inductive invariant automatically with 29 predicates, all of which are simple and do not involve any Boolean connectives. There are 2 index variables in $\mathcal{X}$ to specify the predicates. The abstract reachability took 19 iterations and 263 minutes to produce the inductive invariant [3]. For the simpler version which has single-entry channels for communication, our method finds the inductive invariant in 85s using 17 predicates in 9 iterations. All experiments were performed on a 2.1 GHz Linux machine with 1GB of RAM. The main difficulty of making the channels unbounded is the presence of two-dimensional arrays in the model, and additional state variables for the head and tail pointers for each of the unbounded queues.

For space considerations, we will only describe the nature of predicates used for the model with single-entry channels. A few predicates did not require any index symbol. These include: home_exclusive_granted, home_current_command = REQ_SHARED, home_current_command = REQ_EXCLUSIVE, $\texttt{hd} < \texttt{tl}$ and $\texttt{hd} = \texttt{tl}$. For most predicates, we required a single index variable $\texttt{i} \in \mathcal{X}$, to denote an arbitrary cache index. They include: home_invalidate_list(i), cache(i) = EXCLUSIVE, cache(i) = SHARED, ch_2(i) = GRANT_EXCLUSIVE, ch_2(i) = GRANT_SHARED, ch_2(i) = INVALIDATE, ch_3(i) = INVALIDATE_ACK and $\texttt{i} = \texttt{q}(\texttt{hd})$. We also required another index variable $\texttt{j} \in \mathcal{X}$ to range over the entries of the queue hsl_q. The predicates over j are $\texttt{hd} \leq \texttt{j}$ and $\texttt{j} < \texttt{tl}$. Finally, to relate the entries in hsl_q and hsl_pos, we needed the predicates $\texttt{i} = \texttt{q}(\texttt{j})$ and $\texttt{j} = \texttt{hsl\_pos}(\texttt{i})$.

Most of the predicates are fairly easy to find from the model and from counterexamples. Predicate abstraction constructs an inductive invariant of the form $\forall \texttt{i}, \texttt{j} : \psi^*(\texttt{i}, \texttt{j})$, which implies the cache-coherence property. This implication is checked automatically with a sound decision procedure in UCLID [4], using quantifier instantiation.

Previous attempts at using predicate abstraction (with locally quantified predicates), for a version of the protocol with single-entry channels required complex quantified predicates [7,2], sometimes as complex as an invariant. However, Baukus et al. [2]

[2] The extension was suggested by Steven German himself

[3] There is a lot of scope for optimizing the performance of our procedure.

proved the liveness of the protocol in addition to the cache-coherence property. Pnueli et al. [15] have used the method of *invisible* invariants to derive the inductive invariant for the model with single-entry channels, but it is not clear if their formalism can model the version with unbounded channels per client.

10 Conclusions

We have found quantified invariants to be essential in expressing the properties of systems with function state variables. The ability of our tool to automatically generate quantified invariants based on small and simple predicates allows us to deal with much more complex systems in a more automated fashion than previous work. A next step would be to automatically generate the set of predicates used by the predicate abstraction tool. Other tools generate new predicates based on the counterexample traces from the abstract model [1,7]. This approach cannot be used directly in our context, since our abstract system cannot be viewed as a state transition system, and so there is no way to characterize a counterexample by a single state sequence. We are currently looking at techniques to extract relevant predicates from the proof of unsatisfiable formulas which represent that an error state can't be reached after any finite number of steps.

Acknowledgments. We wish to thank Ching-Tsun Chou for his detailed comments on an early draft of this paper.

References

1. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation (PLDI '01)*, pages 203–213, 2001.
2. K. Baukus, Y. Lakhnech, and K. Stahl. Parameterized Verification of a Cache Coherence Protocol: Safety and Liveness. In A. Cortesi, editor, *Verification, Model Checking, and Abstract Interpretation, (VMCAI '02)*, LNCS 2294, pages 317–330, January 2002.
3. E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem*. Springer-Verlag, 1997.
4. R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In E. Brinksma and K. G. Larsen, editors, *Computer-Aided Verification (CAV '02)*, LNCS 2404, pages 78–92, 2002.
5. C. T. Chou. The mathematical foundation fo symbolic trajectory evaluation. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification (CAV'99)*, LNCS 1633, pages 196–207, 1999.
6. P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (POPL '77)*, pages 238–252, 1977.
7. S. Das and D. L. Dill. Counter-example based predicate discovery in predicate abstraction. In M. D. Aagaard and J. W. O'Leary, editors, *Formal Methods in Computer-Aided Design (FMCAD '02)*, LNCS 2517, pages 19–32, 2002.

8. C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In J. Launchbury and J. C. Mitchell, editors, *Principles of Programming Languages (POPL '02)*, pages 191–202, 2002.
9. S. German. Personal communication.
10. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Computer-Aided Verification (CAV '97)*, LNCS 1254, pages 72–83, 1997.
11. C. N. Ip and D. L. Dill. Verifying systems with replicated components in Murφ. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification (CAV '96)*, LNCS 1102, pages 147–158, 1996.
12. S. K. Lahiri, R. E. Bryant, and B. Cook. A symbolic approach to predicate abstraction. In W. A. Hunt, Jr. and F. Somenzi, editors, *Computer-Aided Verification (CAV '03)*, LNCS 2725, pages 141–153, 2003.
13. S. K. Lahiri, S. A. Seshia, and R. E. Bryant. Modeling and verification of out-of-order microprocessors in UCLID. In M. D. Aagaard and J. W. O'Leary, editors, *Formal Methods in Computer-Aided Design (FMCAD '02)*, LNCS 2517, pages 142–159, 2002.
14. L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17:453–455, August 1974.
15. A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '01)*, LNCS 2031, pages 82–97, 2001.

Analysis of Recursive Game Graphs Using Data Flow Equations

K. Etessami

LFCS, School of Informatics
U. of Edinburgh
kousha@inf.ed.ac.uk

Abstract. Given a finite-state abstraction of a sequential program with potentially recursive procedures and input from the environment, we wish to check statically whether there are input sequences that can drive the system into "bad/good" executions. Pushdown games have been used in recent years for such analyses and there is by now a very rich literature on the subject. (See, e.g., [BS92,Tho95,Wal96,BEM97,Cac02a,CDT02].) In this paper we use recursive game graphs to model such interprocedural control flow in an open system. These models are intimately related to pushdown systems and pushdown games , but more directly capture the control flow graphs of recursive programs ([AEY01,BGR01,ATM03b]).
We describe alternative algorithms for the well-studied problems of determining both reachability and Büchi winning strategies in such games. Our algorithms are based on solutions to second-order data flow equations, generalizing the Datalog rules used in [AEY01] for analysis of recursive state machines. This offers what we feel is a conceptually simpler view of these well-studied problems and provides another example of the close links between the techniques used in program analysis and those of model checking.
There are also some technical advantages to the equational approach. Like the approach of Cachat [Cac02a], our solution avoids the necessarily exponential-space blow-up incurred by Walukiewicz's algorithms for pushdown games. However, unlike [Cac02a], our approach does not rely on a representation of the space of winning configurations of a pushdown graph by (alternating) automata. Only "minimal" sets of exits that can be "forced" need to be maintained, and this provides the potential for greater space efficiency. In a sense, our algorithms can be viewed as an "automaton-free" version of the algorithms of [Cac02a].

1 Introduction

There has been intense activity in recent years aimed at extending the scope of model checking to finite-state abstractions of sequential programs described by modular and potentially recursive procedures. A partial list of references includes [BS92,Wal96,BEM97,Rep98,EHRS00,BR00,AEY01,BGR01,Cac02a,CDT02]. Pushdown systems are one of the primary vehicles for such analyses. When such models are analyzed in the setting of an open system, where the environment

B. Steffen and G. Levi (Eds.): VMCAI 2004, LNCS 2937, pp. 282–296, 2004.

is viewed as an adversary, a natural model to study becomes pushdown games. There is by now a very rich literature on the analysis of pushdown games (see, e.g., [Cau90,BS92,Tho95,Wal96,BEM97,Cac02a,CDT02].)

In this paper we use recursive game graphs to model such interprocedural control flow in an open system. These models are intimately related to pushdown systems and pushdown games, but more directly capture the control flow graphs of recursive programs. Recursive state machines (RSMs) were introduced and studied in [AEY01] and independently in [BGR01], and are related to similar models studied program analysis (see, e.g., [Rep98]). Besides giving algorithms for their analysis, they showed that RSMs are expressively equivalent to pushdown systems, with efficient translations in both directions. More recently [ATM03b,ATM03a] have studied "modular strategies" on Recursive Game Graphs (RGGs), a natural adaptation of RSMs to a game setting. The translations of [AEY01,BGR01] can easily be adapted to show that pushdown games and (labelled) RGGs are expressively equivalent.

The results of Walukiewicz [Wal96] were a key watershed in the analysis of pushdown games. Besides much else, he showed that determining the existence of winning strategies under any parity encodable winning condition is in EXPTIME, and that existence of winning strategies under simple reachability winning conditions is already EXPTIME-hard. In Walukiewicz's algorithm one first constructs from a pushdown game P an exponentially larger (flat) game graph G_P. A winning strategy in G_P corresponds to a winning strategy in P, and one then solves G_P via efficient algorithms for flat games. The disadvantage of such an algorithm from a practical point of view is that it can not get "lucky": exponential space is consumed in the first phase on any input, even if the game P may be very "simple" to solve. Subsequently, many others have studied algorithms for analysis of pushdown systems and pushdown games. One effective approach has been based on the observation that the reaching configurations of a pushdown system form a regular set ([FWW97,BEM97,EHRS00]). This approach has been used more recently by Cachat and others [Cac02a,CDT02,Cac02b,Ser03] to give alternative algorithms for analysis of pushdown games. These algorithms do not necessarily incur the exponential space blow-up incurred by Walukiewicz's algorithm, but they do require construction of an alternating automaton accepting the winning configurations of a pushdown game.

We describe alternative algorithms for determining both reachability and Büchi strategies on RGGs. Our algorithms do not make use of any automata to represent global configurations of the underlying game graph, but are instead based on solutions to second-order data flow equations over sets of exit nodes in the RGGs. This generalizes the Datalog rules used in [AEY01] for analysis of recursive state machines. It is also closely related to the algorithm in [ATM03b] for computing "modular strategies" for reachability on recursive game graphs. Computing modular strategies is NP-complete, whereas computing arbitrary strategies is EXPTIME-complete, so our algorithms necessarily differ. But our underlying ideas for the reachability algorithm are closely related to theirs, and both can be viewed as generalizations of the approach of [AEY01].

The dataflow equation approach offers what we feel to be a conceptually simpler solution to these well-studied problems. It also provides another example of the close links between the techniques used in program analysis and those of model checking. In a sense, our algorithms can be viewed as an "automaton-free" version of the algorithms of [Cac02a], although they arose in an attempt to generalize the approach of [AEY01].

There are some technical advantages to be gained from using the equational approach. Unlike the automaton approach, in which global winning configurations need to be recorded as accepted strings of an alternating automaton, in our approach only "minimal" sets of exits that can be "forced" need to be maintained and this provides the potential for greater space efficiency when one is interested in, e.g., whether particular vertices can be reached under any "context". Also, a formulation based on solutions of data flow equations allows for the application of well established techniques in program analysis for efficient evaluation, such as worklist data structures to manage sets that require updates (see, e.g., [NNH99,App98]).

The sections of the paper are as follows: in section 2 we provide background definitions on recursive game graphs, in section 3 we provide our algorithm for RGGs under a reachability winning condition, in section 4 we extend these to Büchi conditions, and we conclude in 5.

2 Definitions

Syntax. A *recursive game graph (RGG)* A is given by a tuple $\langle A_1, \ldots, A_k \rangle$, where each *component game graph* $A_i = (N_i \cup B_i, Y_i, En_i, Ex_i, \delta_i)$ consists of the following pieces:

- A set N_i of *nodes* and a (disjoint) set B_i of *boxes*.
- A labelling $Y_i : B_i \mapsto \{1, \ldots, k\}$ that assigns to every box an index of one of the component machines, $A_1, \ldots, A_k$.
- A set of *entry* nodes $En_i \subseteq N_i$, and a set of *exit* nodes $Ex_i \subseteq N_i$.
- A transition relation δ_i, where transitions are of the form (u, v) where:
  1. the source u is either a non-exit node in $N_i \setminus Ex_i$, or a pair (b, x), where b is a box in B_i and x is an exit node in Ex_j, where $j = Y_i(b)$;
  2. the destination v is either a non-entry node in $N_i \setminus En_i$, or a pair (b, e), where b is a box in B_i and e is an entry node in En_j, where $j = Y_i(b)$.

Let $N = \bigcup_{i=1}^{k} N_i$ denote the set of all nodes. For a box b in A_i and an entry e of $Y_i(b)$, we define the pair (b, e) to be a *call*. Likewise, for an exit x of $Y_i(b)$, we say (b, x) is a *return*. We will use the term *vertex* to refer collectively to nodes, calls, and returns that participate in some transition, and we denote this set by $Q = \bigcup_{i=1}^{k} Q_i$. That is, the transition relation δ_i is a set of labelled directed edges on the set Q_i of vertices of A_i. Let $\delta = \cup_i \delta_i$ be the set of all edges in A. In addition to the tuple $\langle A_1, \ldots, A_k \rangle$, we are also given a partition of the vertices Q into two disjoint sets Q^0 and Q^1, corresponding to where it is player 0's and player 1's turn to play, respectively.

Semantics. An RGG defines a global game graph $T_A = (V, V_0, V_1, \Delta)$. The global *states* V of T_A, for RGG A, are tuples $\langle b_1, \ldots, b_r, u\rangle$, where $b_1, \ldots, b_r$ are boxes and u is a vertex (not just a node). The global transition relation Δ is given as follows: let $s = \langle b_1, \ldots, b_r, u\rangle$ be a state with u a vertex in Q_j, and $b_r \in B_m$. Then, $(s, s') \in \Delta$ iff one of the following holds:

1. $(u, u') \in \delta_j$ for a vertex u' of A_j, and $s' = \langle b_1, \ldots, b_r, u'\rangle$.
2. u is a call vertex $u = (b', e)$, of A_j, and $s' = \langle b_1, \ldots, b_r, b', e\rangle$.
3. u is an exit-node of A_j, and $s' = \langle b_1, \ldots, b_{r-1}, (b_r, u)\rangle$.

Case 1 corresponds to when control stays within the component A_j, case 2 is when a new component is entered via a box of A_j, case 3 is when the control exits A_j and returns back to A_m.
The global states are partitioned into sets V_0 and V_1, as follows: $s = \langle b_1, \ldots, b_r, u\rangle$ is in V_0 (V_1) if $u \in Q^0$ ($u \in Q^1$, respectively). We augment RGGs with an acceptance condition $\mathcal{F}$. We restrict ourselves to Büchi conditions $F \subseteq N$. The global game graph T_A, together with a start state $init \in N$ and acceptance condition F, define an (infinite) game graph with an acceptance condition, $B_A = (V, V_0, V_1, \Delta, \langle init\rangle, F^\star)$, where $F^* = \{\langle \bar{b}, v\rangle \mid v \in F\}$. B_A defines a game as follows. The game begins at $s_0 = \langle init\rangle$. Thereafter, whenever we are at a state s_i in V_0 (V_1), Player 0 (respectively, Player 1), moves by choosing some transition $(s_i, s_{i+1}) \in \Delta$. A run (or *play*) $\pi = s_0 s_1 \ldots$ is constructed from the infinite sequence of moves. The run π is *accepting* if it is accepted under the Büchi condition $F^\star$, meaning for infinitely many indices i, $s_i \in F^\star$. We say that player 0 wins if π is accepting, and 1 wins if π is not accepting. A player is said to have a *winning strategy* in the Büchi game if it can play in such a way that it will win regardless of how the other player plays (we omit a more formal description of a winning strategy). We will also be interested in simpler reachability winning conditions. Given A, a vertex $u \in Q$, and a set of nodes $Z \subseteq N$, define $u \Rightarrow^0 Z$ to mean that player i has a strategy to reach some state $\langle b_1, \ldots b_r, v\rangle$ such that $v \in Z$, from the state $\langle u\rangle$ in the global game graph T_A. We will be interested in two algorithmic problems:

1. *Winning under reachability conditions:* Given A, a vertex $u \in Q$, and a set of nodes $Z \subseteq N$, determine whether $u \Rightarrow^0 Z$.
2. *Winning under Büchi conditions:* Given A, $init \in N$, and $F \subseteq N$, determine whether one player or the other has a winning strategy in the game defined by B_A under Büchi acceptance conditions.

3 Algorithm for the Reachability Game

To check whether $v \Rightarrow^0 Z$, we will incrementally associate to each vertex v of component A_i a set-of-sets $RSet_Z(v) \subseteq 2^{Ex_i}$ of exit nodes from Ex_i. Usually, when Z is clear from the context, we write $RSet(v)$ instead of $RSet_Z(v)$. The empty set $\{\}$ will eventually end up in $RSet(v)$ iff $v \Rightarrow^0 Z$. We first make a more general definition. For u a vertex in component A_i, and $X \subseteq Ex_i$, and for an

arbitrary set of nodes $Z \subseteq N$, We write $u \Rightarrow^0 (Z, X)$, to mean that player 0 can "force the play", starting at $\langle u \rangle$, to either reach a global state $\langle x \rangle$ for some $x \in X$, or else to reach a global state $\langle b_1, \ldots, b_k, z \rangle$, such that $z \in Z$. Note that $u \Rightarrow^0 Z$ if and only if $u \Rightarrow (Z, \emptyset)$. As we will see, some subset $Y \subseteq X$ will eventually end up in $RSet(u)$ if and only if $u \Rightarrow^0 (Z, X)$. The algorithm to build $RSet(v)$'s proceeds as follows:

1. **Initialization:** for every $z \in Z$, we initialize $RSet(z) := \{\{\}\}$ to the set containing only the empty set. To each exit point $x \in Ex_i$, not in Z, in any component A_i, we associate the set $RSet(x) := \{\{x\}\}$. To all other vertices v we initially associate $RSet(v) := \{\}$.
2. **Rule application:** Inductively, we make sure the relationships defined by rules (a), (b), and (c), below, hold between the sets associated with vertices, based on a standard fixpoint iteration loop. We say an instance of a rule is *applicable* if the right hand side does not equal the left hand side. While there is an applicable instance of a rule, we apply it. For $\mathcal{S} \subseteq 2^D$, over any base set D, let

$$Min(\mathcal{S}) = \{X \in \mathcal{S} \mid \forall\, X' \in \mathcal{S} \text{ if } X' \subseteq X \text{ then } X' = X\}$$

In other words, $Min(\mathcal{S})$ contains the *minimal* sets $X \in \mathcal{S}$ such that there is no $X' \in \mathcal{S}$ which is strictly contained in X.

a) For every 0-vertex $v \in Q^0$ that isn't an exit, isn't a call, and isn't in Z:

$$RSet(v) := Min(\bigcup_{(v,w)\in\delta} RSet(w))$$

In other words, the set of sets associated with v is the union of the set of sets associated with its successors, but only retaining the minimal sets among these. So if it has two successors w_1 and w_2 with $RSet(w_1) = \{\{x_1\}, \{x_2, x_3\}\}$ and $RSet(w_2) = \{\{x_2\}\}$ then $RSet(v) = \{\{x_1\}, \{x_2\}\}$.

b) For every 1-vertex $v \in Q^1$, with successors $\{w_1, ..., w_k\}$, where v is not an exit, isn't a call, and $v \notin Z$:

$$RSet(v) := Min(\{(\bigcup_{i\in\{1,\ldots,k\}} X_i) \mid \forall i : X_i \in RSet(w_i)\})$$

In other words, the set of sets associated with each 1-vertex v will consist of all possible unions of sets of its successors, retaining only minimal sets among these. So, if $RSet(w_1) = \{\{x_1\}, \{x_3\}\}$ and $RSet(w_3) = \{\{x_1, x_2\}, \{x_2, x_3\}\}$, then
$RSet(v) = Min(\{\{x_1, x_2\}, \{x_1, x_2, x_3\}, \{x_2, x_3\}\}) = \{\{x_1, x_2\}, \{x_2, x_3\}\}$.
Note that if for some w_j, $RSet(w_j) = \{\}$, then $RSet(v) := \{\}$.

c) For every call vertex (b, e_i), where e_i is an entry point of A_i, A_i has exit set $Ex_i = \{x_1, ..., x_k\}$, and b is in component A_j, where $Y_j(b) = i$:

$$RSet((b, e_i)) := Min(\{\bigcup_{x\in X} X_{b,x} \mid X \in RSet(e_i) \;\&\; X_{b,x} \in RSet((b, x))\})$$

In other words, the set of sets $RSet((b, e_i)) \subseteq 2^{Ex_j}$ associated with the call (b, e_i) consists of the union, for each set $X \in RSet(e_i) \subseteq 2^{Ex_i}$, of the sets $X_{(b,x)} \in RSet((b, x))$, where $x \in X$, and (b, x) is a return of box b. By convention, if $X = \{\}$, then $\bigcup_{x \in X} X_{b,x} = \{\}$. So empty sets carry over from $RSet(e_i)$ to $RSet((b, e_i))$.

For any two sets-of-sets, $\mathcal{S}, \mathcal{S}' \subseteq 2^D$, over any base set D, we will say that $\mathcal{S}'$ *covers* $\mathcal{S}$ *from below*, and denote this by $\mathcal{S}' \sqsubseteq \mathcal{S}$, iff for every set X in $\mathcal{S}$ there is a set Y in $\mathcal{S}'$ such that $Y \subseteq X$. Note that the empty set $\{\}$ is covered from below by any other set, and that the set $\{\{\}\}$ containing only the empty set covers every set from below. (We omit the lattice-theoretic formulation.)

Theorem 1. *For a vertex u of component A_i, a set $Z \subseteq N$, and a set $X \subseteq Ex_i$:*

1. *Let* $\mathrm{RSet}(u)$ *be the set associated with u at some point during the algorithm, and* $\mathrm{RSet}'(u)$ *be the set associated with u some time later. Then* $\mathrm{RSet}'(u) \sqsubseteq \mathrm{RSet}(u)$.
2. *At any time during the algorithm* $\mathrm{RSet}(u)$ *is minimal, i.e., if* $Y \in \mathrm{RSet}(u)$ *there is no strict subset $Y' \subset Y$, with* $Y' \in \mathrm{RSet}(u)$.
3. *The algorithm will halt, and* $\mathrm{RSet}(u)$ *will get updated at most* $2^{|Ex_i|}$ *times.*
4. *(*) Some subset $Y \subseteq X$ will eventually end up in* $\mathrm{RSet}(u)$ *if and only if (**)* $u \Rightarrow^0 (Z, X)$.

In particular, letting $X = \emptyset$ in (4.)*, $u \Rightarrow^0 Z$ if and only if when the algorithm halts $\{\}$ is in* $\mathrm{RSet}(u)$.

Proof. Claim (1.) asserts a monotonicity property of these rules. Namely, suppose for vertex u, $RSet(u)$ depends on $RSet(w_i)$ for immediate "neighbours" $\{w_1, \ldots, w_k\}$ (if u is a call, these can be either return vertices of the box, or entries of the corresponding component). If we are about to update $RSet(u)$ using the $RSet'$ sets associated with its neighbours, suppose that for some neighbours, $RSet'(w_i) \sqsubseteq RSet(w_i)$, while for others (that haven't been updated) $RSet'(w_i) = RSet(w_i)$. We can show by inspection that applying each rule results in a set $RSet'(u)$ such that $RSet'(u) \sqsubseteq RSet(u)$.
Claim (2.) follows from both the initial settings of $RSet(u)$'s, and the fact that each of our update rules retains only minimal sets.
Claim (3.) follows, because by claims (1.) and (2.) successive sets associated with a vertex will always cover prior ones from below. Hence, since $\sqsubseteq$ defines a partial-order on these sets, there are no non-trivial chains that repeat the same set. Updates are only performed on applicable rule instances, i.e., when the respective set changes. Hence at most $2^{|Ex_i|}$ updates are performed on $RSet(u)$.
Claim (4.): the easier direction is $(*) \rightarrow (**)$: Suppose $Y \subseteq X$, and $Y \in RSet(u)$. By the "soundness" of our rules, there is a way for player 0 to force the game, starting at $\langle u \rangle$, either into a state $\langle y \rangle$ for $y \in Y$, or else into $\langle \bar{b}, z \rangle$, for some $z \in Z$. "Soundness" means: both the initial setting of $RSet(u)$'s, and every rule application preserve the following invariant: if $Y \in RSet(u)$, then $u \Rightarrow^0 (Z, Y)$.
$(**) \rightarrow (*)$: Suppose $u \Rightarrow^0 (Z, X)$. Consider Player 0's winning strategy as a finite tree T_{win} with root $\langle u \rangle$, leaves labelled by two kinds of states, either of the

form $\langle x \rangle$, for $x \in X$, or of the form $\langle b_1, \ldots, b_r, z \rangle$, such that $z \in Z$. An internal 1-state of T_{win} has as its children ALL its neighbours in T_A, while an internal 0-state has a single child, one of its neighbours in T_A. In addition, no non-leaf (internal) node is of the form $\langle b_1, \ldots, b_r, z \rangle$, where $z \in Z$.

For $s = \langle b_1, \ldots, b_r, u \rangle$, let $vertex(s) = u$. Extending the notation, for a set of states S let, $vertex(S) = \{vertex(s) \mid s \in S\}$. Consider a particular occurrence of $s = \langle b_1, \ldots, b_r, u \rangle$ in T_{win}, where $u \in Q_i$.[1] We inductively define a "cut off subtree" T^s_{win} of T_{win}, whose root is (this) s and such that for every state s' of T^s_{win}, if $b_1, \ldots, b_r$ is a prefix of every child of s' (and $vertex(s') \notin Z$) then every child of s' in T_{win} is also a child of s' in T^s_{win}. (Note: either every child of s' has $b_1, \ldots, b_r$ as a prefix, or none does because in that case $s' = \langle b_1, \ldots, b_r, ex \rangle$ and $ex \in Ex_i$). Note that when $s = \langle u \rangle$ is the root of T_{win}, $T^s_{win} = T_{win}$. For a finite strategy tree T, let $leafexits(T) = \{s \mid s \text{ is a leaf of } T, \text{ and } vertex(s) \notin Z\}$. We will show, by induction on the depth of T^s_{win}, that for every state s in T_{win}, $RSet(vertex(s))$ will eventually contain a set $Y \subseteq vertex(leafexits(T^s_{win}))$. Applying this to the root $\langle u \rangle$ of T_{win}, yields that $(**) \rightarrow (*)$.

Base case: if depth of T^s_{win} is 0, then the only state of T^s_{win} is $s = \langle b_1, ..., b_r, u \rangle$, and either $u \in Z$, in which case $RSet(u) := \{\{\}\}$, or else u is an exit node, in which case $RSet(u) := \{\{u\}\}$. In either case $RSet(vertex(s))$ will contain $Y = vertex(leafexits(T^s_{win}))$. Inductively: suppose the depth of T^s_{win} is n, with root $\langle b_1, \ldots, b_r, u \rangle$. Let the children of the root be $s_1, \ldots, s_m$. Each child s_i is itself the root of a cut-off subtree $T^{s_i}_{win}$ of depth $\leq n - 1$ for player 0, and thus by induction for each i, $RSet(vertex(s_i))$ will eventually contain a set $Y_i \subseteq vertex(leafexits(T^{s_i}_{win}))$. We show that $RSet(u)$ will "after one update" contain a $Y \subseteq vertex(leafexits(T^s_{win}))$. There are several cases based on u:

1. u is an exit node. In this case T^s_{win} will always be a trivial 1 node tree. The set $RSet(u)$ will always be the same as its initial setting, and by the same argument as the base case of our induction, $RSet(u)$ will contain the set $vertex(leafexits(T^s_{win}))$.
2. u is in Z. In this case, again, T^s_{win} will always be a trivial 1 done tree. The set $RSet(u)$ will always be $\{\{\}\}$, and so $RSet(u)$ will always be equal to $vertex(leafexits(T^s_{win}))$.
3. u is a non-exit 0-vertex of component A_i. s must have one child s' in the strategy tree T^s_{win}. If $s \rightarrow s'$ is a move of T_A within A_i (i.e., $vertex(s)$ is not a call), then by the inductive claim $RSet(vertex(s'))$ will eventually contain a subset of $vertex(leafexits(T^{s'}_{win}))$. Thus, by rule (a) of the algorithm, $RSet(u)$ will "one update later" also contain a set $Y \subseteq vertex(leafexits(T^{s'}_{win}))$, but since $leafexits(T^s_{win}) = leafexits(T^{s'}_{win})$, we have what we want. If, on the other hand, the move from $s \rightarrow s'$ was a call meaning $e = vertex(s')$ is an entry of A_j, while $u = (b, e)$ is a call, then by induction $RSet(e)$ will eventually contain a $Y \subseteq vertex(leafexits(T^{s'}_{win}))$. Consider this $Y = \{ex_1, \ldots, ex_c\} \subseteq Ex_j$,

[1] There could be multiple occurrences of the same state s of T_A in T_{win}, but we assume that s is being somehow identified uniquely. We could do this by providing, together with s, the specific "path name" to the node s in T_{win}.

and consider $leafexits(T^{s'}_{win}) = \{s_1, \ldots, s_c\}$ in the tree T_{win}. Each s_i has exactly one child s'_i such that $vertex(s'_i) = (b, ex_i)$. Since $T^{s'_i}_{win}$ is a proper subtree of T^{s}_{win}, by induction each $RSet((b, ex_i))$ will eventually contain a set $Y_i \subseteq vertex(leafexits(T^{s'_i}_{win}))$. Now, using the fact that eventually, $RSet(e)$ will contain Y and that each $RSet((b, ex_i))$ will contain Y_i, we can use rule (c) to obtain that "one update later" $RSet((b, e))$ will contain a subset of $\cup_i vertex(leafexits(T^{s'_i}_{win}))$, which itself is a subset of $vertex(leafexits(T^{s}_{win}))$.

4. u is a non-exit 1-vertex of component A_i. In this case, without loss of generality, we can assume the only possibility is that s must have children $s'_1, \ldots, s'_d$ in the strategy tree T_{win}, and all $s \to s'_i$ moves are moves of T_A within A_i (i.e., not a call,, because call moves have only 1 successor in T_A, and hence calls can be viewed as 0-vertices). Then by the inductive claim $RSet(vertex(s'_i))$ will eventually contain a subset Y_i of $vertex(leafexits(T^{s'_i}_{win}))$. Thus, by rule (b) of the algorithm, $RSet(u)$ will "one update later" also contain a set $Y \subseteq \cup_i vertex(leafexits(T^{s'_i}_{win}))$, but since $leafexits(T^{s}_{win}) = \cup_i leafexits(T^{s'}_{win})$, we are done.

□

Let $maxEx = \max_i |Ex_i|$. For an upper bound on the running time of the algorithm, observe that each $RSet(u)$ gets updated at most 2^{maxEx} times. Each rule application can be done in time at most $2^{O(m \cdot maxEx)}$, where m is the maximum number of "neighbouring" vertices v', for any vertex v, such that $RSet(v)$ depends directly on $RSet(v')$ in some rule (the time taken for rule updates can be heavily optimized with good use of data structures common in program analysis). m is clearly upper bounded by the number of vertices, but is typically much smaller. There are $|A|$ vertices, so the worst case running time will be $|A| \cdot 2^{O(m \cdot maxEx)}$. However, observe that this worst-case analysis can be very pessimistic, as the retained minimal sets may converge to a fixpoint well before $RSet(v)$'s ever grow large. That is the principle advantage, and hope, offered by our equational approach. Of course, it will require much experimentation to determine under what circumstances this advantage materializes.

4 Algorithm for the Büchi Case

In the algorithm for Büchi conditions, the set-of-sets $BSet(u)$ that we associate with each vertex will be more elaborate than just subsets of the exits. Recall $maxEx$ is the maximum number of exits of any component in A. Let $Calls$ be the set of calls, i.e., call vertices (b, e), in the RGG. Let $GoodVertices = F \cup Calls$, be the union of the set of accept vertices (nodes) and call vertices in A. Let $maxMeasure = (|GoodVertices| \cdot 4^{maxEx}) + 1$.

Lets briefly sketch the intuition for the algorithm and proof. For each vertex u of A_i, $BSet(u)$ will contain sets of the form: $\{(\bot, m), (ex_1, tval_1), \ldots, (ex_k, tval_k)\}$, where $m \leq maxMeasure$, where each $ex_j \in Ex_i$, and where each $tval_j \in \{true, false\}$. The set can be interpreted to mean that player 0 can play starting at $\langle u \rangle$ in such a way that, no

matter what player 1 does, we either will visit an accept node m times during our play, or else we will reach a state $\langle ex_j \rangle$, and we will do so having visited an accept node along the way if $tval_j = true$. The rules will be used to update these sets in a consistent way. What we will show is that $\{(\bot, maxMeasure)\}$ enters $BSet(u)$ iff there is a finite strategy tree rooted at $\langle u \rangle$ such that on every path in the strategy tree we necessarily repeat a vertex, having visited an accept state in between visits, with the stack getting no smaller in between visits, and such that we are in the same "context of obligations" (to be made precise) in both visits. This allows us to repeatedly apply the same substrategies in order to achieve an infinite winning strategy tree.

Let $P_i = \{(ex, tval) \mid ex \in Ex_i \ \& \ tval \in \{true, false\}\}$. Let $J = \{(\bot, m) \mid 1 \leq m \leq maxMeasure\}$. We say a set $X \in 2^{P_i \cup J}$ is *well-formed* if both the following conditions hold: (1) For all $ex \in Ex_i$, $(ex, true) \notin X$ or $(ex, false) \notin X$ (or both). (2) If $(\bot, m) \in X$ then for all $m' \neq m$, $(\bot, m') \notin X$. Let τ_i be the set of of all well-formed sets in $2^{P_i \cup J}$. We will refer to $X \in \tau_i$ as a set of type τ_i, and similarly we will refer to sets-of-sets $\mathcal{S} \subseteq \tau_i$ as having type τ_i. For $S, S' \in \tau_i$, we write $S' \preceq S$ iff $(a), (b), (c)$ hold: (a) if for $ex \in Ex_i$, $((ex, false) \notin S$ and $(ex, true) \notin S)$, then $((ex, false) \notin S'$ and $(ex, true) \notin S')$. (b) if $(ex, true) \in S$, then $(ex, false) \notin S'$. (c) if $(\bot, j) \in S$ then $(\bot, j') \in S'$ such that $j' \geq j$.
For a set $\mathcal{S} \subseteq \tau_i$ let $Min(\mathcal{S})$ denote the set of those sets $X \in \mathcal{S}$ that are minimal with respect to $\preceq$ in $\mathcal{S}$. Given a set X of type τ_i, let $Increment(X)$ be the smallest set such that: if $(ex, tval)$ is in X, then $(ex, true)$ is in $Increment(X)$, and if $(\bot, j)$ is in X, then $(\bot, min(j+1, maxMeasure))$ is in $Increment(X)$. Extending the notation, for $\mathcal{S} \subseteq \tau_i$, let $Increment(\mathcal{S}) = \{Increment(X) \mid X \in \mathcal{S}\}$.

For sets $X_1, \ldots, X_k$, each of the same type τ, we define a "boxy union" $X' = \bigsqcup_{i \in \{1,\ldots,k\}} X_i$ to be a subset of $X = \bigcup_{i \in \{1,\ldots,k\}} X_i$ as follows: if element $(ex, true)$ and $(ex, false)$ are both in X, then only $(ex, false)$ is in X'. Moreover, there is a $(\bot, j')$ in X' if j' is the minimum value j for which $(\bot, j)$ is in X. Intuitively, "boxy union" reflects choices optimal for the adversary (player 1).

We will associate to each $u \in Q_i$ a set $BSet(u) \subseteq \tau_i$, such that $BSet(u)$ will eventually contain $\{(\bot, maxMeasure)\}$ if and only if player 0 has a winning strategy in the Büchi game. Let F be the set of accepting nodes.
Initialization: For each $z \in F$, initially $\{(\bot, 1)\} \in BSet(z)$. Moreover, for each $x \in Ex_i$, initially $BSet(x)$ contains $\{(x, tval)\}$, where $tval = true$ if $x \in F$, and otherwise $tval = false$. For all other vertices v, we initialize $BSet(v) := \emptyset$.
Rule application: we make sure the following relationships hold:

1. For every 0-vertex v, with the exception of exits or calls, or nodes in F:

$$BSet(v) := Min(\bigcup_{(v,w) \in \delta} BSet(w))$$

2. For every 0-vertex v that isn't a exit or call, but is in F,

$$BSet(v) := Min(Increment(\bigcup_{(v,w) \in \delta} BSet(w)) \cup \{\{(\bot, 1)\}\})$$

3. For every 1-vertex v with successors $\{w_1, ..., w_k\}$ (v not an exit or call), such that v is not in F:

$$BSet(v) := Min(\{ \bigsqcup_{i \in \{1,...,k\}} X_i \mid \forall i : X_i \in BSet(w_i)\})$$

4. For every 1-vertex v with successors $\{w_1, ..., w_k\}$ (v not an exit or call), such that v is in F:

$$BSet(v) := Min(Increment(\{ \bigsqcup_{i \in \{1,...,k\}} X_i \mid \forall i : X_i \in BSet(w_i)\}) \cup \{\{(\bot, 1)\}\})$$

5. For a call (b, e_i) in component A_j, where e_i is an entry point of A_i, and where A_i has exits $\{x_1, ..., x_k\}$,
$BSet((b, e_i)) :=$

$$Min(\{((\bigsqcup_{x \in X} Incr_{X,x}(X_{b,x})) \sqcup D_X) | X \in BSet(e_i) \ \& \ X_{b,x} \in BSet((b, x))\})$$

where $D_X = \{(\bot, j) \mid (\bot, j) \in X\}$, and where $Incr_{X,x}(X')$ is equal to X' if there is an element $(x, false)$ in X, and otherwise is $Increment(X')$.

We call a rule application that changes the value of some $BSet(v)$, an *update*.

Theorem 2. *The algorithm halts after at most* $|A|^2 \cdot 2^{O(maxEx)}$ *updates to each* BSet(v). *When it halt,* $\{(\bot, maxMeasure)\} \in$ BSet(u) *if and only if player* 0 *has a winning strategy in the Büchi game* B_A *starting at* $\langle u \rangle$.

For the proof please see the appendix. For the worst-case time complexity: each update can be carried out in time at most $|A|^{O(1)} \cdot 2^{O(m \cdot maxEx)}$, where m is the maximum number of "neighbouring" vertices of any vertex. So the total running time is $|A|^{O(1)} \cdot 2^{O(m \cdot maxEx)}$. The algorithm as described will always require at least $maxMeasure$ updates to reach $\{(\bot, maxMeasure)\}$ in some $BSet(v)$. The algorithm can be reformulated to avoid this. We omit such a reformulation.

5 Conclusions

We have provided alternative algorithms, using second-order data flow equations, for determining whether a player has a winning strategy on recursive game graphs, a model that is expressively equivalent to pushdown games. Our algorithms generalize the approach of using Datalog rules for analysis of recursive state machines from [AEY01], as in [ATM03b], and they can also be viewed as a "automaton-free" version of the algorithms given by [Cac02a] for pushdown games. Several extensions of Cachat's work have appeared in more recent literature. [CDT02] extends the algorithms to check properties such as "stack boundedness" of infinite plays (a notion which was studied for runs of recursive state machines in [AEY01]). Also, [Ser03,Cac02b] extends the work to games with parity conditions. It may be possible to carry out these extensions in our equational framework, but we have not done so here.

Acknowledgements. Thanks to R. Alur, P. Madhusudan, and M. Yannakakis for extensive discussions and helpful comments.

References

[AEY01] R. Alur, K. Etessami, and M. Yannakakis. Analysis of recursive state machines. In *Proc. of CAV'01*, pp. 304–313, 2001.

[App98] A. Appel. *Modern compiler implementation*. Cambridge U. Press, 1998.

[ATM03a] R. Alur, S. La Torre, and P. Madhusudan. Modular strategies for infinite games on recursive graphs. In *CAV'03*, *LNCS* vol. 2725, pages 67–79, 2003.

[ATM03b] R. Alur, S. La Torre, and P. Madhusudan. Modular strategies for recursive game graphs. In *TACAS'03*, *LNCS* vol. 2619 , pp. 363–378, 2003.

[BEM97] A. Boujjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: App's to model checking. In *CONCUR'97*, pages 135–150, 1997.

[BGR01] M. Benedikt, P. Godefroid, and T. Reps. Model checking of unrestricted hierarchical state machines. In *ICALP'01*, *LNCS* 2076, pp. 652–666, 2001.

[BR00] T. Ball and S. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN'2000*, volume 1885 of *LNCS*, pages 113–130, 2000.

[BS92] O. Burkart and B. Steffen. Model checking of context-free processes. In *Proc. of CONCUR*, volume 836 of *LNCS*, pages 123–137, 1992.

[Cac02a] T. Cachat. Symbolic strategy synthesis for games on pushdown graphs. In *Proc. of ICALP*, volume 2380 of *LNCS*, 2002.

[Cac02b] T. Cachat. Uniform solution of parity games on prefix recognizable graphs. In *Infinity 2002, 4th. Int. Workshop*, 2002.

[Cau90] D. Caucal. On the regular structure of prefix rewriting. In *5th Coll. on Trees in Algebra and Programming*, *LNCS* vol. 431, pp. 87–102, 1990.

[CDT02] T. Cachat, J. Duparc, and W. Thomas. Solving pushdown games with a $sigma_3$ winning condition. In *CSL'02*, pp. 322–336, 2002.

[EHRS00] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *12th Computer-Aided Verification*, volume 1855 of *LNCS*, pages 232–247. Springer, 2000.

[FWW97] A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. In *Infinity'97 Workshop*, volume 9 of *Electronic Notes in Theoretical Computer Science*, 1997.

[NNH99] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[Rep98] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12):701–726, 1998.

[Ser03] O. Serre. Note on winning strategies on pushdown games with omega-regular winning conditions. *Information Processing Letters*, 85(6):285–291, 2003.

[Tho95] W. Thomas. On the synthesis of strategies in infinite games. In *STACS*, volume 900 of *LNCS*, pages 1–13, 1995.

[Wal96] I. Walukiewicz. Pushdown processes: games and model checking. In *Computer-Aided Verification*, pages 62–75, 1996.

A Proof of Correctness for Büchi Case

Proof. (of Theorem 2) Due to space we state several lemmas without proof. First, ($\Rightarrow$). We show that if $\{(\bot, maxMeasure)\}$ ends up in $BSet(u)$ then Player 0 has a winning strategy in the game (this is the harder direction). Suppose $S = \{(\bot, maxMeasure)\}$ does end up in $BSet(u)$. Our first step is to construct a "witness" to this in the form of a straight-line program, W, using the sets $BSet(v)$. A witness W for u has the form: $(1, C_1, Pre_1)(2, C_2, Pre_2)\ldots(l, C_l, Pre_1)$. It consists of l *lines* of the form (d, C_d, Pre_d), with d the *line number*, and:

- C_d is the *content*, and has the form (v, S) where $v \in Q_i$ is a vertex of A_i, $S \in \tau_i$, and there exists $S' \in BSet(v)$ such that $S' \preceq S$.
  If $(\bot, m) \in S$, we say line d "contains a measure", and its measure is m.
- Pre_d is a (possibly empty) list $[d_1, \ldots, d_k]$ of predecessor line numbers, with $d_i < d$ for each i.

Morever:

- No two lines have the same content, thus content determines line number.
- A line with no predecessors is called an *initial* line. If the d'th line is initial, then it must be either of the form: $(d, (v, \{(\bot, 1)\}), [])$, where $v \in F$, or of the form $(d, (ex, \{(ex, tval)\}), [])$ where ex is an exit, and $tval$ is true if and only if $ex \in F$.
- If the d'th line has predecessors, it has the form $(d, (v, S), [d_1, \ldots, d_i])$, where:
  - The content (v, S) is *implied in one step* by the content of its predecessors. By this we mean that if the contents of the d_j are, respectively, (v_j, S_j), then there is a rule $\mathcal{R}$ such that if $S_j \in BSet(v_j)$ for each j, then one application of $\mathcal{R}$ would put $S' \in BSet(v)$, where $S' \preceq S$.
    E.g., if $v = (b, e)$ is a call, and $S = \{(\bot, m), (ex_1, tval_1), \ldots, (ex_k, tval_k)\}$, then d_1 will be a line with content (e, S_1) and d_j, for $j > 1$, will have content $((b, ex'_j), S_j)$, such that $(ex'_j, tval)$ is in S_1, moreover so that $S = \bigsqcup_{j \in \{2,\ldots,i\}} S_j \sqcup \{(\bot, m') \mid (\bot, m') \in S_1\}$.
  - Moreover, the measures $(\bot, m')$ in any of the content sets $S_1, \ldots, S_i$, of lines $d_1, \ldots, d_i$, should be as weak as possible in order to imply S, meaning that it should not be possible to decrease the measure in any single S_j and still imply S in one step.
- The last line, line l, is of the form $(l, (u, \{(\bot, maxMeasure)\}), Pre_l)$.

Lemma 1. *If* $\{(\bot, maxMeasure)\} \in \mathrm{BSet}(u)$ *then a witness* W *for* u *exists.*

The lemma can be proved by induction on the number of rule applications it took for $\{(\bot, maxMeasure)\}$ to enter $BSet(u)$. We will use W to define a winning strategy tree for player 0. To do this, we will first need some facts about W. By a *path* from a line d to a line d', $d > d'$, in W, we mean a sequence $\gamma = d_1, \ldots, d_n$, such that $d = d_1$, $d' = d_n$, and $d_{i+1} \in Pre_{d_i}$ for all $i \in \{1, \ldots, n-1\}$. (Note: the path goes from higher to lower line numbers.)

Lemma 2. *On any path $\gamma = d_1, \ldots, d_n$ in W, the measure is non-increasing, meaning, for $i < j$, if line d_i contains a measure m then line d_j either contains a measure $m' \leq m$, or does not contain a measure, and if line d_i does not contain a measure then neither does line d_j.*

Proof. A line has a measure iff any of its predecessors has a measure. Moreover, because of the minimality constraint on predecessor measures in W, the measure of a line $(d, (v, S), Pre_d)$ is the same as that of its predecessors that contain a measure unless v is either a call or an accept node (i.e., a good vertex), in which case the measure can be 1 greater. The measure is therefore non-increasing. □

For a path γ in W, let $goodLines(\gamma)$ be the number of lines of γ of the form $(d, (v, S), Pre_d)$ where $v \in GoodVertices$, i.e., v is either a call vertex or an accept node. Let d and d' be two lines such that there is a path from d to d' in W, and let $goodDist_W(d, d') = \min\{goodLines(\gamma) \mid \gamma \text{ is a path from } d \text{ to } d' \text{ in } W\}$. For two lines $(d, (v, S), Pre_d)$ and $(d', (v', S'), Pre_{d'})$ that both contain a measure, we say the contents C_d and C'_d are *the same except for the measure* if $v = v'$, and S and S' differ only by the fact that $(\bot, m) \in S$, and $(\bot, m') \in S'$, with $m \neq m'$. In such a case, we write $C_d \ll C'_d$, if $m < m'$.

Lemma 3. *Let d be any initial line of W of the form $(d, (v, \{(\bot, 1)\}), [])$ and let l be the last line number, then*

1. $\text{goodDist}_W(l, d) \geq maxMeasure$.
2. *On any path $\gamma = l = d_r d_{r-1} \ldots d_1 = d$ from l to d, there exist two distinct lines d_i and d_j, $i > j$, such that $C_{d_j} \ll C_{d_i}$.*

Proof. (1) The measure at l is $maxMeasure$, while the measure at d is 1. Thus, in a path γ from line l to d, since we must have $maxMeasure - 1$ opportunities to decrement the measure, and can only do so on good lines, and since line d is also a good line, we must encounter at least $maxMeasure$ good lines.
(2) There are at most $(|GoodVertices| \cdot 4^{|maxEnt|})$ different contents (v, S) where v is a good-vertex (counting only once contents that the same except for the measure). Thus, since $maxMeasure = (|GoodVertices| \cdot 4^{|maxEnt|}) + 1$, by the pigeon-hole principle and by (1), there must be two such lines d_i and d_j in any γ. Since the measure is non-increasing, we must have $C_{d_j} \ll C_{d_i}$. □

We now use W to construct a strategy tree for player 0. Each node of the strategy tree will be labelled by a triple $(s, d, Stack)$, where:

- s is a global state $\langle \bar{b}, v \rangle$ of B_A.
- d is a line number in the witness straight-line program W.
- $Stack$ is a stack $[\beta_j, \beta_{j-1}, \ldots, \beta_1]$, where each β_r defines a mapping from a subset of the exits of a component to line numbers in W (in a consistent way to be defined).

The root of the strategy tree will be labelled by $(\langle u \rangle, l, [])$, (where l is the last line of W). Thereafter, if a node is labelled by $(\langle \bar{b}, v \rangle, d, Stack)$, we use W to construct its children. Suppose, for example, line d of W is $(d, (v, S), [d_1, d_2, \ldots, d_i])$, and

suppose $v = (b, e)$ is a call. Let line d_1 have content (e, S_1). Then we create one child for the node and label it by the tuple $(\langle \bar{b}, b, e\rangle, d_1, push(\beta, Stack))$, where β is a mapping from each ex_j, such that $(ex_j, tval_j)$ is in S_1, to a line d_{i_j} whose content is of the form $((b, ex_j), S_j)$. In other words, the element pushed on the Stack tells us where in the witness program to return to when returning from b on the call stack, as dictated by the predecessors $d_2, \ldots, d_i$ in line d. For other kinds of vertices v, we can construct corresponding children of nodes in the strategy tree. If in the strategy tree so constructed we reach a state $\langle \bar{b}, b, ex\rangle$, whose vertex ex is an exit, then we pop *Stack* and use its contents to dictate what the children of that node should be labelled, using the state content to assign the line number of W to the triple associated with the children.

Lemma 4. *The construction outlined above yields a finite tree T_W, all of whose leaves are labelled by triples of form $(\langle \bar{b}, v\rangle, d, Stack)$, where line d is an initial line of W with content $(v, \{(\bot, 1)\})$.*

The lemma can be established using the structure of W. The key point is that if we ever reach an exit following the program W, we know that our Stack contains a return address where we may continue to build T_W. We want to build from T_W an infinite strategy tree.

Lemma 5. *For any leaf z in T_W labelled by $(\langle \bar{b}, v\rangle, d, Stack)$, the root-to-leaf path in T_W must include a subsequence $H_{maxMeasure}, \ldots, H_1$, where $H_i = (\langle \overline{b_i}, v_i\rangle, d_i, Stack_i)$, and where*

- *v_i is a good vertex of A, for each i,*
- *each $\overline{b_i}$ is a prefix of $\bar{b}$, and every node on the path from H_i to z contains $\overline{b_i}$ as a prefix of its call stack (and hence the Stack at every such node also contains as a prefix the stack $Stack_i$ at H_i).*
- *H_i has (v_i, S_i) as the content of its line, such that $(\bot, i) \in S_i$.*

In other words, the subsequence $H_{maxMeasure}, \ldots, H_1$ of nodes along the root to leaf path in T_W will witness the decrementation of the counter from $maxMeasure$ down to 1. The counter can only be decremented by 1 at a time, and so all such distinct witnesses must exist. Now, because of the size of $maxMeasure$, there must be two distinct H_r and $H_{r'}$, $r' < r$, with labels $(\langle \overline{b_r}, v\rangle, d_r, Stack_r)$ and $(\langle \overline{b_{r'}}, v\rangle, d_{r'}, Stack_{r'})$, such that they both have the same vertex v (which must be a good vertex, either an accept node or a call), and such that the lines d_r and $d_{r'}$ have content (v, S) and (v, S'), where S and S' are exactly the same except that $(\bot, r)$ is in S while $(\bot, r')$ is in S'. We mark any such node $H_{r'}$. We then eliminate the subtrees rooted at all marked nodes in T_W. This yields another finite tree T'. We will use T' to construct an infinite strategy tree $T^\star$ that is accepting for player 0. Let M be the set of (good) vertices v such that a state $\langle \bar{b}, v\rangle$ labels some leaf of T'.

Lemma 6. *For any leaf of T' labelled $L = (\langle \bar{b}, v\rangle, d, Stack)$, there is a finite strategy tree T_L for player 0, with root labelled L, such that the state labels on every root-to-leaf path contain $\bar{b}$ as a stack prefix, and such that all leaf labels $(\langle \overline{b'}, v'\rangle, d', Stack')$, have $v' \in M$. Moreover, every root-to-leaf path in T_L contains an accept state.*

Using the lemma we can incrementally construct $T^\star$ from T'. We repeatedly attaching a copy of T_L to every leaf labelled L in the tree T'. Since the process always produces leaves labelled s whose node v is in M, and since the stacks can only grow, we can extend the tree indefinitely. Every path will contain infinitely many accept states, because each finite subtree that we attach contains an accept state on every root to leaf path.

($\Leftarrow$): If Player 0 has a winning strategy in the Büchi game starting at $\langle u \rangle$, then $\{(\bot, maxMeasure)\}$ will eventually enter $BSet(u)$. We omit the proof, which is similar to the proof that if player 0 has a winning strategy in the reachability game from $\langle u \rangle$, then $\{\}$ will eventually enter $RSet(u)$. □

Applying Jlint to Space Exploration Software

Cyrille Artho[1] and Klaus Havelund[2]

[1] Computer Systems Institute, ETH Zurich, Switzerland
[2] Kestrel Technology, NASA Ames Research Center, Moffett Field, California USA

Abstract. Java is a very successful programming language which is also becoming widespread in embedded systems, where software correctness is critical. Jlint is a simple but highly efficient static analyzer that checks a Java program for several common errors, such as `null` pointer exceptions, and overflow errors. It also includes checks for multi-threading problems, such as deadlocks and data races. The case study described here shows the effectiveness of Jlint in finding certain faults, including multi-threading problems. Analyzing the reasons for false positives in the multi-threading warnings gives an insight into design patterns commonly used in multi-threaded code. The results show that a few analysis techniques are sufficient to avoid almost all false positives. These techniques include investigating all possible callers and a few code idioms. Verifying the correct application of these patterns is still crucial, because their correct usage is not trivial.

1 Introduction

Java is becoming more widespread in the area of embedded systems, both as a scaled-down "Micro Edition" [20] or by having real-time extensions [6,5]. In such systems, software cannot always be replaced on a running system. Failures may have expensive or even catastrophic consequences. These costs are obviously prohibitively high when a software-related problem causes the failure of a space craft [14]. Therefore an automated tool which can detect faults easily, preferably early in the lifecycle of software, can be very useful. One tool that allows fault detection easily, even in incomplete systems, is Jlint. Among similar tools geared towards Java, it is one of the most suitable with respect to ease of use (no annotations required) and free availability (the tool is Open Source) [1].

1.1 The Java Programming Language

Java is a modern, object-oriented programming language that has had a large success in the past few years. Source code is not compiled to machine code, but to a different form, the *bytecode.* This bytecode runs in a dedicated environment, the *virtual machine.* In order to guarantee the integrity of the system, each class file containing bytecode is checked prior to execution [11,19,21].

The Java language allows each object to have any number of *fields,* which are attributes of each object. These may be static, i.e., shared among all instances of a certain class, or dynamic, i.e., each instance has its own fields. In contrast to that, *local variables* are thread-local and only visible within one method.

B. Steffen and G. Levi (Eds.): VMCAI 2004, LNCS 2937, pp. 297–308, 2004.

Java allows inheritance: a method of a given class may be *overridden* by a method of the same name. Similarly, fields in a subclass *shadow* those with the same name in the superclass. In general, these mechanisms work well for small code examples but may be dangerous in larger projects. Methods overriding other methods must ensure they do not violate invariants of the superclass. Similar problems occur with variable shadowing. The programmer is not always aware that a variable with the same name already exists on a different level, such as the superclass.

In order to prevent incorrect programs from corrupting the system, Java's virtual machine has various safety mechanisms built in. Each variable access is guarded against manipulating memory outside the allocated area. In particular, pointers must not be `null` when dereferenced, and array indices must be in a valid range. If these properties are violated, an *exception* is thrown indicating a programming error. This is a highly undesirable behavior in most cases. Ideally, such errors should be prevented by static analysis, rather than caught at run-time.

Furthermore, Java offers mechanisms to write multi-threaded programs. The two key mechanisms are locking primitives, using the `synchronized` keyword, and inter-thread synchronization with the `wait` and `notify` methods. Incorrect lock usage using too many locks may lead to *deadlocks.* For example, if two threads each wait on a lock held by the other thread, both threads cannot continue their execution. On the other hand, if a value is accessed with insufficient lock protection, *data races* may occur: two threads may access the same value concurrently, and the results of the operations are no longer deterministic.

Java's message passing mechanisms for threads also is a source of problems. A call to `wait` allows a thread to suspend until a condition becomes true, which must be signaled by `notify` by another thread. When calling `wait` the calling thread must ensure that it owns the lock it waits on, and also release any other locks before the call. Otherwise, remaining locks held are unavailable to other threads, which may in turn block when trying to obtain them. This can prevent them from calling `notify` which would allow the waiting thread to release its lock. This situation is also a deadlock.

1.2 Related Work

Much effort has gone into fault-finding in Java programs, single-threaded and multi-threaded. The approaches can be separated into *static checkers,* which check a program at compile-time and try to approximate its run-time behavior, and *dynamic checkers,* which try to catch and analyze anomalies during program execution.

Several static analysis tools exist that examine a program for faults such as `null` pointer dereferences or data races. The ESC/Java [9] tool is, like Jlint, also based on static analysis, or more generally on theorem proving. It, however, requires annotation of the program. While it is more precise than Jlint, it is not nearly as fast and requires a large effort from the user to fully exploit the power of this tool [9].

Dynamic tools have the advantage of having more precise information available in the execution trace. The Eraser algorithm [22], which has been implemented in the Visual Threads tool [12] to analyze C and C++ programs, is such an algorithm that examines a program execution trace for locking patterns and variable accesses in order to predict potential data races.

The Java PathExplorer tool (JPaX) [16] performs deadlock analysis and the Eraser data race analysis on Java programs. It furthermore recently has been extended with the high-level data race detection algorithm described in [3]. This algorithm analyzes how collections of variables are accessed by multiple threads.

More heavyweight dynamic approaches include model checking, which explores all possible schedules in a program. Recently, model checkers have been developed that apply directly to programs (instead of just models thereof). This includes the Java PathFinder system (JPF) developed by NASA [15,24], and similar systems [10,8,17,4, 23]. Such systems, however, suffer from the state space explosion problem. In [13] we describe an extension of Java PathFinder which performs data race analysis (and deadlock analysis) in simulation mode, whereafter the model checker is used to demonstrate whether the data race (deadlock) warnings are real or not.

This paper focuses on applying Jlint [2] to the software for detecting errors statically. Jlint uses static analysis and abstract interpretation to find difficult errors at compile-time. A similar case study with Jlint has been made before, applying it to large projects [2]. The difference to this case study is that the other case study had scalability in mind. Jlint had been applied to packages containing several hundred thousand lines of code, generating hundreds of warning messages. Because of this, the warnings had been evaluated selectively, omitting some hard-to-check deadlock warnings. In this case study, an effort was made to analyze every single warning and also see what kinds of design patterns cause false positives.[1]

1.3 Outline

This text is organized as follows: Section 2 describes Jlint and how it was used for this project. Sections 3 and 4 show the results of applying Jlint to space exploration program code. Design patterns which are common among these two projects are analyzed in Section 5. Section 6 summarizes the results and concludes.

2 Jlint

2.1 Tool Description

Jlint checks Java code and finds bugs, inconsistencies and synchronization problems by performing a data flow analysis, abstract interpretation, and building the lock graph. It issues warnings about potential problems. These warnings do not imply that an actual error exists. This makes Jlint unsound as a program prover. Moreover, Jlint can also miss errors, making it incomplete. The reason for this is that the goal was to make Jlint practical, scalable, and possible to implement it in a short time.

Typical warnings about possible faults issued by Jlint are `null` pointer dereferences, array bounds overflows, and value overflows. The latter may occur if one multiplies two 32 bit integer values without converting them to 64 bit first.

[1] Design patterns commonly denote compositions of objects in software. In this paper, the notion of composition is different. It includes lock patterns and sometimes only applies to a small part of the program. In that context, we also use the term “code idiom”.

Many warnings that Jlint issues are code guidelines: A local variable should never have the same name as a field of the same class or a superclass. When a method of a given name is overridden, all its variants should be overridden, in order to guarantee a consistent behavior of the subclass.

Jlint also includes many analyses for multi-threaded programs. Some of Jlint's warnings for multi-threaded programs are overly cautious. For instance, possible data race warnings for method calls or variable accesses do not necessarily imply a data race. The reason for such false positives are both difficulties inherent to static analysis, such as pointer aliasing across method calls, and limitations in Jlint itself, where its algorithms could be refined with known techniques.

Jlint works in two passes: a first pass, where all methods are analyzed in a modular way, and a second pass with the deadlock analysis. In the first pass, each method is analyzed with abstract interpretation. The abstraction used for numbers includes their maximal possible range and, for integers, bit masks that apply to them. For pointers, the abstraction records whether a pointer is possibly `null` or not, distinguishing the special `this` pointer, values loaded from fields, and new instance references created by the object constructor. The data flow analysis merges possible program states at branch targets but only executes loops once.

Most properties, such as possible value overflows, are analyzed in this first pass. The lock graph is also built in this first pass and checked for deadlocks in a second pass. A possible refinement would be to defer some data race analyses to the second pass, where global information, such as if a field is read-only, can be made available.

2.2 Warning Review Process

Jlint gives fairly descriptive warnings for each problem found. The context given is limited to the class in which the error occurs, the line number, and fields used or methods called. This is always sufficient to find the source of simple warnings, which concern *sequential properties* such as `null` pointer dereferences. These warnings are easy to review and were considered in a first review phase. The other warnings, concerning multi-threading problems, take much more time to consider, and were evaluated in a second phase.

The review process essentially checks whether the problems described in the warnings can actually occur at run-time. In simple cases, warnings may be ruled out given the algorithmic properties of the program. Complex cases include reviewing callers to the method in question.

Data race and deadlock warnings fall in the latter category. They require constructing a part of the call graph including locks owned by callers when a method is called. If it can be ensured that all calls to non-synchronized, shared methods are made only through methods that already employ lock protection, then there cannot be a data race.[2]

This review process can be rather time-consuming. Many warnings occur in similar contexts, so warnings referring to the same problem can usually be easily confirmed as duplicates. This part of the review process was not yet automated in any way but could

[2] Methods that access a shared field are also considered "shared" in this context. The lock used for ensuring mutual exclusion must be the same lock for all calls.

be automated to a large extent with known techniques. Both cases studies were made without prior knowledge of the program code. It can be assumed that the time to review the warnings is shorter for the author of the code, especially when reviewing data race or deadlock warnings.

During the review process, Jlint's warnings were categorized to see whether they refer to the same problem. Such situations constitute calls to the same method from different callers, the same variable used in different contexts, or the same design pattern applied throughout the class. In a separate count, counting the number of distinct problems rather than individual warnings, all such cases were counted once. Note that the review activity was often interrupted by other activities such as writing this paper. We believe this reduced the overall time required because manual code reviews require much attention, and cannot be carried out in one run without a degradation of the concentration required.

3 First Case Study: Rover Code

The first case study is a software module, called the Executive, for controlling the movement of the planetary wheeled rover K9, developed at NASA Ames Research Center. The run time for analyzing the code with Jlint was 0.10 seconds on a PowerPC G4 with a clock frequency of 500 MHz.

3.1 Description of the Rover Project

K9 is a hardware platform for experimenting with rover technology for exploration of the Martian surface. The Executive is a software module for controlling the rover, and is essentially an interpreter of plans, where a plan is a special form of a program. Plans are constructed from high-level constructs, such as sequential composition and conditionals, but no while loops. The effect of while loops is achieved by assuming that plans are generated on the fly during rover operation as environment conditions change. The lowest level nodes of a plan are tasks to be directly executed by the rover hardware. A node in a plan can be further constrained by a set of conditions, which when failing during execution, cause the Executive to abort the execution of the subsequent sibling nodes, unless specified otherwise through options. Examples of conditions are pre-conditions and post-conditions, as well as invariants to be maintained during the execution of the node. The examined Executive consists of 7,300 lines of Java code. This code was extracted by a colleague from the original rover code, written in 35,000 lines of C++. The code is highly multi-threaded, and hence provides a risk for concurrency errors. The Java version of the code was extracted as part of a different project, the purpose of which was to compare various formal methods, such as model checking, static analysis, runtime analysis, and simple testing [7]. The code contained a number seeded of errors.

3.2 Jlint Evaluation

Jlint issues 249 warnings when checking the Rover code. Table 1 summarizes Jlint's output. The first two columns show each type of problem and how many warnings Jlint generated for them. The third, forth and fifth column show the result of the manual

Table 1. Jlint's warnings for the Rover code.

Type	Warnings	Problems found	Correct warnings	False positives	Time [min.]
`null` pointer	5	1	4	1	10
Integer overflow	2	2	2	0	5
`equals` overridden but not `hashCode`	2	1	2	0	1
String comparison as reference	1	0	0	1	1
Total: Sequential errors	10	4	8	2	17
Incorrect `wait/notify` usage	21	5	5	16	26
Data race, method call	157	5	18	139	112
Data race, field access	31	0	0	31	43
Deadlock	30	7	20	10	36
Total: Multi-threading errors	239	17	43	196	217
Total	249	21	51	198	234

source code analysis: how many actual, distinct faults, or at least serious problems, in the code were found, how many warnings described such actual faults, and how many were considered to be false positives. The last column shows the time spent on code review. In the first phase, focusing on sequential properties, ten warnings were reviewed, while the second phase had 239 warnings to be reviewed.

Sequential errors: Among the problems found are two integer overflows, where two 32-bit integers were multiplied to produce a 64 bit result. However, integer conversion took place *after* the 32 bit multiplication, where an overflow may occur.

Two other warnings referred to one problem, where `equals` was overridden, but not `hashCode`. This is dangerous because the modified `equals` method may return true for comparing two objects even though their `hashCode` differs, which is forbidden [21].

A noteworthy false positive concerned two strings that were compared as references. This was correct in that context because one of the strings was always known to be `null`.

Multi-threading errors: The number of deadlock and data race warnings given by Jlint was almost prohibitive. Yet, for answering the question why the false positives were generated, all warnings were investigated. All warnings were relatively easy to analyze. In most cases, possible callers were within the same class. Only for the most complex class, the call graph was large, making analysis more difficult.[3]

A surprisingly high number of multi-threading warnings were of type "Method `'<this>.wait|notify|notifyAll'` is called without synchronizing on `'<this>'`." After discounting dead code and false positives, one scenario remained: A lock was obtained conditionally, although it should be obtained in *all* cases, as required by the Java semantics for `wait` and `notify`. In the Rover code, this reflects a global switch in the original C++ program that would allow testing the program without locking, eliminating

[3] The portion of the call graph to be investigated for this was up to eight methods deep.

possible deadlocks at the cost of introducing data races. Java does not allow this, so the Java version of the program always needs to be run with locking enabled.

All data race warnings about shared field accesses were false positives. Reasons for false positives include the use of thread-local copies [18] or a thread-safe container class. In one case, only one thread instance that could access the shared field is ever generated. Evaluating data races for method calls was even more difficult and time-consuming. The errors found referred to cases where a read-only pattern, was broken by certain methods, creating potential data races. Because of their high number, the distribution of method data race warnings is noteworthy. A few classes which embody parallelized algorithms incurred the largest number of warnings, which were also the hardest to review. Classes encapsulating data are usually much simpler. Because some of these were heavily used in the program, a few of them were also responsible for a large number of warnings. However, these warnings were usually much easier to review.

The 30 deadlock warnings all referred to the same two classes. There were two sets of warnings, the first set containing ten, the second one 20 warnings. The first ten warnings, all of them false positives, showed incomplete synchronization loops in the lock graph. The next 20 warnings, referring to seven methods, showed the same ten warnings with another edge in the lock graph, from the callee class back to the caller. Such a synchronization loop includes two sequences of different lock acquisitions in reverse order. This makes a cyclic deadlock possible. Therefore these warnings referred to actual faults in the code.

Results: In only 15 minutes, four faults could be found by looking at the ten warnings referring to sequential properties. While reviewing the multi-threading warnings was time-consuming due to the complex interactions in the code, it helped to highlight the critical parts of the code. The effort was justifiable for a project of this complexity.

3.3 Comparison to Other Projects

In an internal case study at NASA Ames [7], several other tools were applied to the Rover code base, detecting 38 errors. Among these errors were 18 seeded faults. Interestingly, most of these errors found were not those detected by Jlint. Almost all the seeded bugs concerned algorithmic problems or hard-to-find deadlocks, which Jlint was not capable of finding. However, Jlint in turn detected a lot of faults which were not found by any other tool. Table 2 compares Jlint to the other case studies. In that table, missed faults include both sequential and multi-threading properties.

The eleven new bugs found by Jlint were a great success, even considering that the seven deadlocks correspond to two classes where other deadlocks have been known to occur. However, Jlint reported different methods than those reported in other analyses.

4 DS1

The second case study consisted of an attitude control system and a fault protection system for the Deep Space 1 (DS1) space craft. It took 0.17 seconds to check the entire code base on the same PowerPC G4 with a clock frequency of 500 MHz.

Table 2. Comparison of errors found by Jlint and by other tools.

Error type	#	Evaluation
Seeded faults	18	Not found by Jlint
Non-seeded faults, other than overflow	18	Not found by Jlint
Integer overflow	2	Found by both case studies
`null` pointer	**1**	**New** (i.e., only found by Jlint)
`equals` overridden but not `hashCode`	1	Translation artifact (not occurring in the C version)
Incorrect `wait`/`notify` usage	5	Debugging artifact (not executable in Java)
Data races	5	**3 new**, 2 dead code (unused methods)
Deadlocks	**7**	**new** (two classes known to be faulty involved)

4.1 Description of DS1

DS1 was a technology-testing mission, which was launched October 24 1998, and which ended its primary mission in September 1999. DS1 contained and tested twelve new kinds of space-travel technologies, for example, ion propulsion and artificial intelligence for autonomous control. DS1 also contained more standard technologies, such as an attitude-control system and a fault-protection system, coded in C. The attitude-control system monitors and controls the space craft's attitude, that is, its position in 3-dimensional space. The attitude is controlled by small thrusters, which can be pointed, and fired, in different directions. The fault-protection system monitors the operation of the space craft and initiates corrective actions in case errors occur. The code examined in this case study is an 8,700-line Java version of the attitude-control system and fault-protection system, created in order to examine the potential for programming flight software in Java, as described in [5]. That effort consisted in particular of experimenting with the real-time specification for Java [6]. The original C code was re-designed in Java, using best practices in object-oriented design. The Java version used design patterns extensively, and put an emphasis on pluggable technology, relying on interfaces.

4.2 Jlint Evaluation

Sequential errors: Again, a first evaluation of Jlint's warnings included only the sequential cases. Table 3 shows an overview. Eleven warnings referred to name clashes in variable names, a large risk of future programming errors. False positives resulted from either dead code, a code idiom that was poor choice but acceptable in that case, and compiler artifacts introduced by inner classes. Three warnings reported problems with overridden methods, where several versions of a method with the same name but different parameter lists ("signatures") were only partially overridden. This must be avoided because inconsistencies among the overridden and inherited variants are almost inevitable.

Multi-threading errors: In the second phase, the 47 multi-threading warnings were investigated. Most of them were false positives: Warnings about `run` methods which are

Table 3. Jlint's warnings for the DS1 code.

Type	Warnings	Problems found	Correct warnings	False positives	Time [min.]
Local variable shadows field	4	2	2	2	2
Component shadows base class	7	0	0	7	3
Incomplete method overriding	3	3	3	0	3
`equals` overridden but not `hashCode`	1	0	0	1	1
Total: Sequential errors	15	5	5	10	9
Incorrect `wait/notify` usage	7	0	0	7	3
`run` method not `synchronized`	5	0	0	5	0
Overriding `synchronized` methods	3	0	0	3	2
Data race, field access	1	0	0	1	7
Data race, method call	20	1	6	14	38
Deadlock	11	0	0	11	20
Total: Multi-threading errors	47	1	6	41	70
Total	62	6	11	51	79

not `synchronized` are overly conservative. Warnings about `wait/notify` were caused by the unsoundness of Jlint's data flow analysis. False positives for data race warnings were mostly caused by the fact that Jlint does not analyze all callers when checking methods for thread safety. If all callers synchronize on the same lock, a seemingly unsafe method becomes safe. Other reasons for false positives were the use of thread-safe container classes in such methods, the use of read-only fields, and per-thread confinement [18], which always creates a new instance as return value.

The six warnings indicating an error concerned calls to a logger method. In the logger method, there were indeed data races, even though they may not be considered to be crucial: The output of different formatting elements of different entries to be logged may be interleaved.

Again, as in all non-trivial examples, deadlock warnings are almost impossible to investigate in detail without a call graph browsing tool. Nevertheless, an effort was made. After 12 minutes, it was found that the first deadlock warning was a false alarm due to the lack of context sensitivity in Jlint's call graph analysis. After this, most warnings could be dismissed as duplicates of the first one. In the two remaining cases, Jlint's warnings did not give the full loop context, so they could not be used.

Results: Most sequential warnings could be evaluated very quickly. The problems found were code convention violations, which would not necessarily cause run-time errors. However, they are easy to fix and should be addressed. Reviewing the data race warnings was relatively simple, although it would have been much easier with a call graph visualization tool. Most false positives could have been prevented by a more complete call graph analysis or recognizing a few simple design patterns.

Table 4. Design patterns for avoiding data races in seemingly unsafe methods.

Code base	Rover			DS1		Total
Problem category	`wait/notify`	Data race (field)	Data race (method)	Data race (field)	Data race (method)	
Read-only fields	–	9	19	–	3	31
Synchronization for all callers	12	–	7	1	2	22
Return copy of data	–	–	3	–	1	4
Thread-local copy during operation	–	1	1	–	–	2
Thread-safe container	–	–	1	–	2	3
One thread instance	–	1	–	–	–	1
Total	12	11	31	1	8	63

5 Design Patterns in Multi-threaded Software

Sections 3 and 4 have shown that sequential properties are easy to evaluate with the aid of a static analysis tool. This is not the case with multi-threading problems. There are two ways to improve the situation: Make the evaluation of warnings easier using visualization tools, or improve the quality of the analysis itself, reducing the false positives. We focused on the latter aspect. When analyzing the warnings, it soon became apparent that only a few common code idioms were behind the problems. The remainder of this paper investigates what patterns are used to avoid multi-threading problems.

Table 4 shows an overview of the different design patterns used in the code of the two space exploration projects to avoid conflicts with unprotected fields or methods. The counts correspond to the applications of these patterns, all of which result in one or more spurious warnings when analyzed with Jlint. When using these patterns, there appears to be a data race, if a method is considered in isolation or without considering thread ownership. There is no data race when considering the entire program.

The most common idiom used to prevent data races was the use of read-only values. Read-only fields are usually declared `final` and not changed after initialization. Because this declaration discipline is not always followed strictly, recognizing it statically is not always trivial, but nevertheless feasible by checking all uses of a given field in the entire code. Ensuring global thread-safety in such cases is of course only possible in the absence of dynamic class loading. Other design patterns include:

- Ensuring mutual exclusion in an unsafe method by having all callers of that method acquire a common lock. Such callers work as a thread-safe wrapper around unsynchronized parts of the code.
- The usage of (deep) copies of data returned by a method ensures that the "working copy" used subsequently by the caller remains thread-local [18]. This eliminates the need for synchronization in the caller.
- Copying method parameters restricts data ownership to the called method and the current thread [18]. The callee then does not have to be synchronized, but it is not allowed to use any shared data other than the copied parameters supplied by the caller. Doing otherwise would again require synchronization.

- Legacy container data structures such as `Vector` are inherently thread-safe because they internally use synchronization [21].
- Finally, if there exists only one thread instance of a particular class, no data races can occur if that thread is the only type that calls a certain method.

Two cases of false positives were not included in this summary: unused methods (dead code) and conditional locking based on a global flag used for debugging `wait/notify` locking (which was permissible in the original C++ Rover code but not in the Java version).

This study indicates that four design patterns prevail in cases where code is apparently not thread-safe: Synchronization of all callers, use of read-only values, thread-local copies of data, and the use of thread-safe container classes. Although simple patterns prevail, their usage is not always trivial: Some of the data race warnings for the Rover code pointed out cases where it was attempted to use the read-only pattern, but the use was not carried out consistently throughout the project. Such a small mistake violates the property that guarantees thread-safety. This discussion so far concerned only data race warnings. No prevailing pattern has been found in the case of deadlocks, where the programmer has to ensure no cyclic lock dependency arises between threads.

6 Conclusions

Space exploration software is complex. The high costs incurred by potential software failures make the application of fault-finding tools very fruitful. Jlint was very successful as such a tool in both case studies, complementing the strengths of other tools. In each project, the study found four or five significant problems within only 15 minutes of evaluating Jlint's warnings. The multi-threading warnings were more difficult and time-consuming to evaluate but still effective at pointing out critical parts in the code.

An analysis of the false positives showed that in apparently thread-unsafe code, four common design patterns ensure thread-safety in all cases. Static analysis tools should therefore be extended with specific algorithms geared towards these patterns to reduce false positives. Furthermore, these patterns were not always applied correctly and are still a significant source of programming errors. This calls for tools that verify the correct application of these patterns, thereby pointing out even more subtle errors than previously possible.

References

1. C. Artho. Finding Faults in Multi-Threaded Programs. Master's thesis, ETH Zürich, 2001.
2. C. Artho and A. Biere. Applying Static Analysis to Large-Scale, Multi-threaded Java Programs. In D. Grant, editor, *Proceedings of the 13th ASWEC*, pages 68–75. IEEE CS Press, 2001.
3. C. Artho, K. Havelund, and A. Biere. High-Level Data Races. In *VVEIS'03*, April 2003.
4. T. Ball, A. Podelski, and S. Rajamani. Boolean and Cartesian Abstractions for Model Checking C Programs. In *Proc. TACAS'01: Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, Italy, 2001.

5. E. G. Benowitz and A. F. Niessner. Java for Flight Software. In *Space Mission Challenges for Information Technology*, July 2003.
6. G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.
7. G. Brat, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, A. Venet, and W. Visser. Experimental Evaluation of Verification and Validation Tools on Martian Rover Software. In *SEI Software Model Checking Workshop*, 2003. Extended abstract.
8. J. Corbett, M. B. Dwyer, J. Hatcliff, C. S. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting Finite-state Models from Java Source Code. In *Proc. 22nd International Conference on Software Engineering*, Ireland, 2000. ACM Press.
9. D. L. Detlefs, K. Rustan, M. Leino, G. Nelson, and J. B. Saxe. Extended Static Checking. Technical Report 159, Compaq Systems Research Center, Palo Alto, California, USA, 1998.
10. P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, pages 174–186, France, 1997.
11. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Virtual Language Specification, Second Edition*. Addison Wesley, 2000.
12. J. Harrow. Runtime Checking of Multithreaded Applications with Visual Threads. In *7th SPIN Workshop*, volume 1885 of *LNCS*, pages 331–342. Springer, 2000.
13. K. Havelund. Using Runtime Analysis to Guide Model Checking of Java Programs. In *7th SPIN Workshop*, volume 1885 of *LNCS*, pages 245–264. Springer, 2000.
14. K. Havelund, M. Lowry, S. Park, C. Pecheur, J. Penix, W. Visser, and J. White. Formal Analysis of the Remote Agent Before and After Flight. In *5th NASA Langley Formal Methods Workshop*, June 2000. USA.
15. K. Havelund and T. Pressburger. Model Checking Java Programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.
16. K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. In K. Havelund and G. Roşu, editors, *Runtime Verification (RV'01)*, volume 55 of *ENTCS*. Elsevier Science, 2001.
17. G. Holzmann and M. Smith. A Practical Method for Verifying Event-Driven Software. In *Proc. ICSE'99, International Conference on Software Engineering*, USA, 1999. IEEE/ACM.
18. D. Lea. *Concurrent Programming in Java, Second Edition*. Addison Wesley, 1999.
19. T. Lindholm and A. Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison Wesley, 1999.
20. Sun Microsystems. Connected, limited device configuration. specification version 1.0a, may 2000. `http://java.sun.com/j2me/docs/`.
21. Sun Microsystems. Java 2 documentation. `http://java.sun.com/j2se/1.4/docs/`.
22. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.
23. S. D. Stoller. Model-Checking Multi-threaded Distributed Java Programs. In *7th SPIN Workshop*, volume 1885 of *LNCS*, pages 224–244. Springer, 2000.
24. W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *Proc. ASE'2000: The 15th IEEE International Conference on Automated Software Engineering*. IEEE CS Press, 2000.

Why AI + ILP Is Good for WCET, but MC Is Not, Nor ILP Alone

Reinhard Wilhelm

Informatik
Universität des Saarlandes
D - 66123 Saarbrücken

Abstract. A combination of Abstract Interpretation (AI) with Integer Linear Programming (ILP) has been successfully used to determine precise upper bounds on the execution times of real-time programs, commonly called worst-case execution times (WCET). The task solved by abstract interpretation is to verify as many local safety properties as possible, safety properties who correspond to the absence of "timing accidents". Timing accidents, e.g. cache misses, are reasons for the increase of the execution time of an individual instruction in an execution state. This article attempts to give the answer to the frequently encountered claim, "one could have done it by Model Checking (MC)!". It shows that it is the characteristic property of abstract interpretation, which proves AI to be applicable and successful, namely that it only needs one fixpoint iteration to compute invariants that allow the derivation of many safety properties. MC seems to encounter an exponential state-space explosion when faced with the same problem. ILP alone has also been used to model a processor architecture and a program whose upper bounds for execution times was to be determined. It is argued why the only ILP-only approach found in the literature has not led to success.

1 Introduction

Hard real-time systems are subject to stringent timing constraints which are dictated by the surrounding physical environment. A schedulability analysis has to be performed in order to guarantee that all timing constraints will be met ("timing validation"). Existing techniques for schedulability analysis require upper bounds for the execution times of all the system's tasks to be known. These upper bounds are commonly called the *worst-case execution times* (WCETs) and we will stick to this doubtful naming convention. In analogy, lower bounds on the execution time have been named *best-case execution times*, (BCET). WCETs (and BCETs) have to be *safe*, i.e., they must never underestimate (overestimate) the real execution time. Furthermore, they should be *tight*, i.e., the overestimation should be as small as possible.

Note, that customers of WCET tools want to see how far the WCET is away from the deadline. They are not content with a YES/NO answer, "your program will always terminate (cannot be guaranteed to terminate) within the given deadline". Often, customers even want to see BCETs to see how big the

B. Steffen and G. Levi (Eds.): VMCAI 2004, LNCS 2937, pp. 309–322, 2004.

interval is. This gives them a feeling for the quality of the approximation. Also, any schedulability analysis uses WCETs to exploit the difference between the deadlines and the WCETs as leeway in the scheduling process.

For processors with fixed execution times for each instruction methods to compute sharp WCETs [18,17] have long been established. However, in modern microprocessor architectures caches, pipelines, and all kinds of speculation are key features for improving performance. Caches are used to bridge the gap between processor speed and the access time of main memory. Pipelines enable acceleration by overlapping the executions of different instructions. The consequence is that the execution time of individual instructions, and thus the contribution of one execution of an instruction to the program's execution time can vary widely. The interval of execution times for one instruction is bounded by the execution times of the following two cases:

- The instruction goes "smoothly" through the pipeline; all loads hit the cache, no pipeline hazard happens, i.e., all operands are ready, no resource conflicts with other currently executing instructions exist.
- "Everything goes wrong", i.e., instruction and operand fetches miss the cache, resources needed by the instruction are occupied, etc.

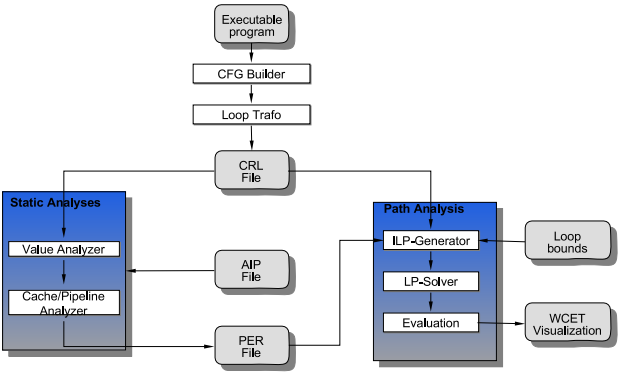


Fig. 1. The architecture of the `aiT` WCET tool.

We will call any increase in execution time during an individual instruction's execution a *timing accident* and the amount of increase the *timing penalty* of this accident. Our experience shows that added timing penalties can be in the order of several hundred processor cycles. The execution time of an instruction depends on the execution state, e.g., the contents of the cache(s), the occupancy of other resources, and thus on the execution history. It is therefore obvious that the execution time cannot be determined in isolation from the execution history.

A more or less standard generic architecture for WCET tools has emerged [8, 4,22]. Fig. 1 shows one instance of this architecture. A first pass, depicted on the left, predicts the behaviour of processor components for the instructions of the program. This allows to derive safe upper bounds for the execution times of

basic blocks. A second pass, the column on the right, computes an upper bound on the execution times over all possible paths of the program.

A combination of Abstract Interpretation–for the first pass–and Integer Linear Programming–for the second pass–has been successfully used to determine precise upper bounds on the execution times of real-time programs running on processors used in embedded systems [24,9,5,6,1]. A commercially available tool, `aiT` by AbsInt, cf. `http://www.absint.de/wcet.htm`, was implemented and is used in the aeronautics and automotive industries.

The contribution of an individual instruction to the total execution time of a program may vary widely depending on the execution history. There are in general too many possible execution histories as to regard them exhaustively. Different execution histories may be summarized to *contexts*, if their influence on the behaviour of instructions or sequences of instructions does not significantly differ. Experience has shown [24], that the right differentiation of control-flow contexts is decisive for the precision of the prediction [15]. Contexts are associated with *basic blocks*, i.e., maximally long straight-line code sequences that can be only entered at the first instruction and left at the last. They indicate through which sequence of function calls and loop iterations control arrived at the basic block.

2 Problem Statement

First the *problem statement*: Given are

- an (abstract) processor model. This model needs to be conservative with respect to the timing properties of the processor, i.e., any duration of a program execution derived from the abstract model must not be shorter than the duration of the program execution on the real processor. For this reason, we call this abstract processor model the *timing model* of the processor. Different degrees of abstraction are possible. The more abstract the timing model is, the more efficient could the analysis be.
- a fully linked executable program. This is the program whose WCET should be determined. It is annotated with all necessary information to make WCET determination possible, e.g. loop iteration counts, maximal recursion depths etc.
- possibly also a deadline, within which the program should terminate. Formally, it makes a difference for model-checking approaches, whether we have the deadline or not. In practice, it does not make a difference.

A *solution* consists in an upper bound for the execution times of all executions of the program

3 WCET Determination by an AI + ILP Approach

3.1 Modularization

Methods based on AI and ILP are realized in the standard architecture mentioned in the Introduction. They have (at least) two phases:

1. Predict safely the processor behaviour for the given program. This is done by computing invariants about the processor state at each program point, such as the set of memory blocks that surely are in the cache every time execution reaches this program point, as well as the contents of prefetch queues and free resources for the corresponding instructions. This information allows the derivation of *safety properties*, namely the absence of timing accidents. Examples of such safety properties are *instruction fetch will not cause a cache miss*; *the load pipeline stage will not encounter a data hazard*.
   Any such safety property allows to reduce the estimated execution time of an instruction by the corresponding timing penalty. Actually, for efficiency reasons upper bounds are calculated for basic blocks.
2. Determine one path through the program, on which the upper bound for the execution times is computed. This is done by mapping the control flow of the program to an integer linear program maximizing execution time for paths through the program and solving this ILP [21,23].

3.2 Cache-Behaviour Prediction

How Abstract Interpretation is used to compute invariants about cache contents is described next. How the behavior of programs on processor pipelines is predicted can be found elsewhere [10,20,25].

Cache Memories. A cache can be characterized by three major parameters:

- *capacity* is the number of bytes it may contain.
- *line size* (also called block size) is the number of contiguous bytes that are transferred from memory on a cache miss. The cache can hold at most $n =$ *capacity*/*line size* blocks.
- *associativity* is the number of cache locations where a particular block may reside.
  $n/$*associativity* is the number of *sets* of a cache.

If a block can reside in any cache location, then the cache is called *fully associative*. If a block can reside in exactly one location, then it is called *direct mapped*. If a block can reside in exactly A locations, then the cache is called A*-way set associative*. The fully associative and the direct mapped caches are special cases of the A-way set associative cache where $A = n$ and $A = 1$, resp.

In the case of an associative cache, a cache line has to be selected for replacement when the cache is full and the processor requests further data. This is done according to a *replacement strategy*. Common strategies are *LRU* (Least Recently Used), *FIFO* (First In First Out), and *random*.

The set where a memory block may reside in the cache is uniquely determined by the address of the memory block, i.e., the behavior of the sets is independent of each other. The behavior of an A-way set associative cache is completely described by the behavior of its n/A fully associative sets. This holds also for direct mapped caches where $A = 1$.

For the sake of space, we restrict our description to the semantics of fully associative caches with LRU replacement strategy. More complete descriptions that explicitly describe direct mapped and A-way set associative caches can be found in [7,6].

Cache Semantics. In the following, we consider a (fully associative) cache as a set of cache lines $L = \{l_1, \ldots, l_n\}$ and the store as a set of memory blocks $S = \{s_1, \ldots, s_m\}$.
To indicate the absence of any memory block in a cache line, we introduce a new element I; $S' = S \cup \{I\}$.

Definition 1 (concrete cache state).
A *(concrete) cache state* is a function $c : L \to S'$.
C_c denotes the set of all concrete cache states. The initial cache state c_I maps all cache lines to I.

If $c(l_i) = s_y$ for a concrete cache state c, then i is the relative age of the memory block according to the LRU replacement strategy and not necessarily the physical position in the cache hardware.

The *update* function describes the effect on the cache of referencing a block in memory. The referenced memory block s_x moves into l_1 if it was in the cache already. All memory blocks in the cache that had been more recently used than s_x increase their relative age by one, i.e., they are shifted by one position to the next cache line. If the referenced memory block was not yet in the cache, it is loaded into l_1 after all memory blocks in the cache have been shifted and the 'oldest', i.e., least recently used memory block, has been removed from the cache if the cache was full.

Definition 2 (cache update). A cache update function $\mathcal{U} : C_c \times S \to C_c$ determines the new cache state for a given cache state and a referenced memory block.

Updates of fully associative caches with LRU replacement strategy are pictured as in Figure 2.

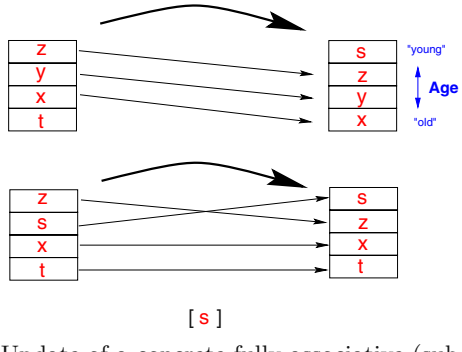


Fig. 2. Update of a concrete fully associative (sub-) cache.

Control Flow Representation. We represent programs by control flow graphs consisting of nodes and typed edges. The nodes represent *basic blocks*. A basic block is a sequence (of fragments) of instructions in which control flow enters at the beginning and leaves at the end without halt or possibility of branching except at the end. For cache analysis, it is most convenient to have one

memory reference per control flow node. Therefore, our nodes may represent the different fragments of machine instructions that access memory. For non-precisely determined addresses of data references, one can use a set of possibly referenced memory blocks. We assume that for each basic block, the sequence of references to memory is known (This is appropriate for instruction caches and can be too restricted for data caches and combined caches. See [7,1] for weaker restrictions.), i.e., there exists a mapping from control flow nodes to sequences of memory blocks: $\mathcal{L} : V \to S^*$.

We can describe the effect of such a sequence on a cache with the help of the update function $\mathcal{U}$. Therefore, we extend $\mathcal{U}$ to sequences of memory references by sequential composition: $\mathcal{U}(c, \langle s_{x_1}, \ldots, s_{x_y} \rangle) = \mathcal{U}(\ldots(\mathcal{U}(c, s_{x_1}))\ldots, s_{x_y})$.

The cache state for a path $(k_1, \ldots, k_p)$ in the control flow graph is given by applying $\mathcal{U}$ to the initial cache state c_I and the concatenation of all sequences of memory references along the path: $\mathcal{U}(c_I, \mathcal{L}(k_1).\ \ldots\ .\mathcal{L}(k_p))$.

The *Collecting Semantics* of a program gathers at each program point the set of all execution states, which the program may encounter at this point during some execution. A semantics on which to base a cache analysis has to model cache contents as part of the execution state. One could thus compute the collecting semantics and project the execution states onto their cache components to obtain the set of all possible cache contents for a given program point. However, the collecting semantics is in general not computable.

Instead, one restricts the standard semantics to only those program constructs, which involve the cache, i.e., memory references. Only they have an effect on the cache modelled by the cache update function, $\mathcal{U}$. This coarser semantics may execute program paths which are not executable in the start semantics. Therefore, the *Collecting Cache Semantics* of a program computes a superset of the set of all concrete cache states occurring at each program point.

Definition 3 (Collecting Cache Semantics).
The Collecting Cache Semantics *of a program is*

$$C_{coll}(p) = \{\mathcal{U}(c_I, \mathcal{L}(k_1).\ \ldots\ .\mathcal{L}(k_n)) \mid (k_1, \ldots, k_n) \textit{ path in the CFG leading to } p\}$$

This collecting semantics would be computable, although often of enormous size. Therefore, another step abstracts it into a compact representation, so called abstract cache states. Note that every information drawn from the abstract cache states allows to safely deduce information about sets of concrete cache states, i.e., only precision may be reduced in this two step process. Correctness is guaranteed.

Abstract Semantics. The specification of a program analysis consists of the specification of an abstract domain and of the abstract semantic functions, mostly called *transfer functions*. The least upper bound operator of the domain combines information when control flow merges.

We present two analyses. The *must analysis* determines a set of memory blocks that are in the cache at a given program point whenever execution reaches this point. The *may analysis* determines all memory blocks that may be in the cache at a given program point. The latter analysis is used to determine the absence of a memory block in the cache.

Table 1. Categorizations of memory references and memory blocks.

Category	Abb.	Meaning
always hit	ah	The memory reference will always result in a cache hit.
always miss	am	The memory reference will always result in a cache miss.
not classified	nc	The memory reference could neither be classified as ah nor am.

The analyses are used to compute a categorization for each memory reference describing its cache behavior. The categories are described in Table 1.

The domains for our abstract interpretations consist of *abstract cache states*:

Definition 4 (abstract cache state). An *abstract cache state* $\hat{c} : L \to 2^S$ maps cache lines to sets of memory blocks. $\hat{C}$ denotes the set of all abstract cache states.

The position of a line in an abstract cache will, as in the case of concrete caches, denote the relative age of the corresponding memory blocks. Note, however, that the domains of abstract cache states will have different partial orders and that the interpretation of abstract cache states will be different in the different analyses.

The following functions relate concrete and abstract domains. An *extraction function*, *extr*, maps a concrete cache state to an abstract cache state. The *abstraction function*, *abstr*, maps sets of concrete cache states to their best representation in the domain of abstract cache states. It is induced by the extraction function. The *concretization function*, *concr*, maps an abstract cache state to the set of all concrete cache states represented by it. It allows to interpret abstract cache states. It is often induced by the abstraction function, cf. [16].

Definition 5 (extraction, abstraction, concretization functions). *The* extraction function $extr : C_c \to \hat{C}$ *forms singleton sets from the images of the concrete cache states it is applied to, i.e.,* $extr(c)(l_i) = \{s_x\}$ *if* $c(l_i) = s_x$.

The abstraction function $abstr : 2^{C_c} \to \hat{C}$ *is defined by*

$$abstr(C) = \bigsqcup \{extr(c) \mid c \in C\}$$

The concretization function $concr : \hat{C} \to 2^{C_c}$ *is defined by*

$$concr(\hat{c}) = \{c \mid extr(c) \sqsubseteq \hat{c}\}$$

So much of commonalities of all the domains to be designed. Note, that all the constructions are parameterized in $\sqcup$ and $\sqsubseteq$.

The transfer functions, the *abstract cache update* functions, all denoted $\hat{\mathcal{U}}$, will describe the effects of a control flow node on an element of the abstract domain. They will be composed of two parts,

1. "refreshing" the accessed memory block, i.e., inserting it into the youngest cache line,
2. "aging" some or all other memory blocks already in the abstract cache.

Termination of the analyses. There are only a finite number of cache lines and for each program a finite number of memory blocks. This means, that the domain of abstract cache states $\hat{c} : L \to 2^S$ is finite. Hence, every ascending chain is finite. Additionally, the abstract cache update functions, $\hat{\mathcal{U}}$, are monotonic. This guarantees that all the analyses will terminate.

Must Analysis. As explained above, the must analysis determines a set of memory blocks that are in the cache at a given program point whenever execution reaches this point. Good information is the knowledge that a memory block is in this set. The bigger the set, the better. As we will see, additional information will even tell how long it will at least stay in the cache. This is connected to the "age" of a memory block. Therefore, the partial order on the *must*–domain is as follows. Take an abstract cache state $\hat{c}$. Above $\hat{c}$ in the domain, i.e., less precise, are states where memory blocks from $\hat{c}$ are either missing or are older than in $\hat{c}$. Therefore, the $\sqcup$-operator applied to two abstract cache states $\hat{c}_1$ and $\hat{c}_2$ will produce a state $\hat{c}$ containing only those memory blocks contained in both and will give them the maximum of their ages in $\hat{c}_1$ and $\hat{c}_2$ (see Figure 4). The positions of the memory blocks in the abstract cache state are thus the upper bounds of the *ages* of the memory blocks in the concrete caches occurring in the collecting cache semantics. Concretization of an abstract cache state, $\hat{c}$, produces the set of all concrete cache states, which contain all the memory blocks contained in $\hat{c}$ with ages not older than in $\hat{c}$. Cache lines not filled by these are filled with other memory blocks.
We use the abstract cache update function depicted in Figure 3.

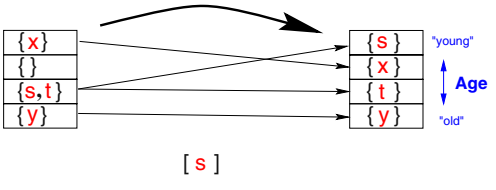


Fig. 3. Update of an abstract fully associative (sub-) cache.

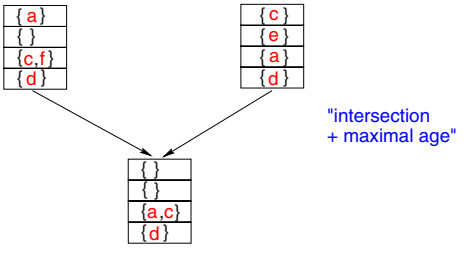


Fig. 4. Combination for must analysis

The solution of the must analysis problem is interpreted as follows: Let $\hat{c}$ be an abstract cache state at some program point. If $s_x \in \hat{c}(l_i)$ for a cache line l_i then s_x will definitely be in the cache whenever execution reaches this program point. A reference to s_x is categorized as *always hit* (ah). There is even a stronger interpretation of the fact that $s_x \in \hat{c}(l_i)$. s_x will stay in the cache at least for the next $n - i$ references to memory blocks that are not in the cache or are *older* than the memory blocks in $\hat{c}$, whereby s_a is older than s_b means: $\exists l_i, l_j : s_a \in \hat{c}(l_i), s_b \in \hat{c}(l_j), i > j$.

May Analysis. To determine, if a memory block s_x will never be in the cache, we compute the complimentary information, i.e., sets of memory blocks that *may* be in the cache. "Good" information is that a memory block is not in this set, because this memory block can be classified as definitely not in the cache whenever execution reaches the given program point. Thus, the smaller the sets are, the better. Additionally, the older blocks will reach the desired situation to be removed from the cache faster than the younger ones. Therefore, the partial order on this domain is as follows. Take some abstract cache state $\hat{c}$. Above $\hat{c}$ in the domain, i.e., less precise, are those states which contain additional memory blocks or where memory blocks from $\hat{c}$ are younger than in $\hat{c}$. Therefore, the $\sqcup$-operator applied to two abstract cache states $\hat{c}_1$ and $\hat{c}_2$ will produce a state $\hat{c}$ containing those memory blocks contained in $\hat{c}_1$ or $\hat{c}_2$ and will give them the minimum of their ages in $\hat{c}_1$ and $\hat{c}_2$ (see Figure 5).

The positions of the memory blocks in the abstract cache state are thus the lower bounds of the *ages* of the memory blocks in the concrete caches occurring in the collecting cache semantics.

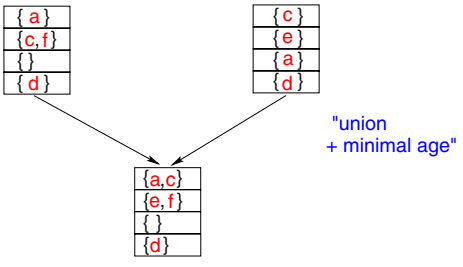


Fig. 5. Combination for may analysis

The solution of the may analysis problem is interpreted as follows: The fact that s_x is in the abstract cache $\hat{c}$ means, that s_x may be in the cache during some execution when the program point is reached. We can even say, that memory block $s_x \in \hat{c}(l_i)$ will surely be removed from the cache after at most $n - i + 1$ references to memory blocks that are not in the cache or are *older or of the same age* than s_x, if there are no memory references to s_x. s_a is older than or of the same age as s_b means: $\exists l_i, l_j : s_a \in \hat{c}(l_i), s_b \in \hat{c}(l_j), i \geq j$. If s_x is not in $\hat{c}(l_i)$ for any l_i then it will definitely be not in the cache on any execution. A reference to s_x is categorized as *always miss* (am).

3.3 Worst Case Path Analysis Using Integer Linear Programming (ILP)

The structure of a program and the set of program paths can be mapped to an ILP in a very natural way. A set of constraints describes the control flow of the program. Solving these constraints yields very precise results [22]. However, requirements for precision of the analysis results demands regarding basic blocks in different contexts, i.e., in different ways, how control reached them. This makes the control quite complex, so that the mapping to an ILP may be very difficult [23].

A problem formulated in an ILP consists of two parts: the cost function and constraints on the variables used in the cost function. Our cost function represents the number of CPU cycles. Correspondingly, it has to be maximised. Each variable in the cost function represents the execution count of one basic block of the program and is weighted by the execution time of that basic block. Additionally, variables are used corresponding to the traversal counts of the edges in the control flow graph.

The integer constraints describing how often basic blocks are executed relative to each other can be automatically generated from the control flow graph. However, additional information about the program provided by the user is usually needed, as the problem of finding the worst case program path is unsolvable in the general case. Loop and recursion bounds cannot always be inferred automatically and must therefore be provided by the user.

The ILP approach for program path analysis has the advantage that users are able to describe in precise terms virtually anything they know about the program. In our system, arbitrary integer constraints can be added by the user to improve the analysis. The system first generates the obvious constraints automatically and then adds user supplied constraints to tighten the WCET bounds.

4 WCET Determination by MC

Each abstract interpretation has encoded in it the type of invariants it is able to derive. It analyzes given programs and derives invariants at each of their program points, called *local* invariants. These invariants then imply safety properties of the execution states at these program points, analogously called *local* safety properties. In our application, we are interested in the verification of the largest subset of a set of local safety properties, that could hold at a program point. The fundamental question is, *what on the MC side corresponds to this ability of verifying an a priori unknown set of local safety properties.* The following analogy comes to one's mind. P. Cousot has shown in [3] that the well-known Constant Propagation analysis cannot be done by model checking. A constant-propagation analysis determines at each program point a set of variable-constant pairs (x, c) such that x is guaranteed to have value c each time execution reaches this program point. The model checker would need to know the constant value c of program variable x in order to derive that x has value c at program point n whenever execution reaches n.

Here are two alternatives, how WCETs could be determined by MC. They all use exponential search to compute upper bounds instead of YES/NO answers in the following way:
A measured execution time is taken as intital value for an upwards search doubling the expected bound each time, until the MC verifies that the bound suffices. A consecutive binary search determines an upper bound. This process needs a logarithmic number of runs of the model checker. An added value of this procedure may be that it delivers two values enclosing the WCET at convergence.

Let us regard the question raised above under two different definitions of model checking:

Model checking is fixpoint iteration without dynamic abstraction and using set union to collect states. This coincides with the view that *model checking is Reachability Analysis.* In contrast, *abstract interpretation* would be considered as *fixpoint iteration with dynamic abstraction using lattice join to combine abstract states.*

This would, however, need to search an exponential state space. Let us regard again the subproblem of cache-behavior prediction. In Section 3, it was described how abstract cache states compactly represent information about sets of concrete cache states. Practice shows that little precision is lost by the abstraction. On the other hand, for an a-way set-associative cache the number of contents of one set is large, namely $\binom{k}{a}\, a!$ if k is the number of memory blocks mapped to this set.

Model checking checks whether a given (global) property holds for a finite transition system. As explained above, we are interested in verifying as many local safety properties at program points as possible. Hence, we either do $O(n)$ runs of the model checker for a program of size n with a possible large constant stemming from the number of potential safety properties, or we code these all together into one $O(n)$-size conjunction.

5 WCET Determination by ILP

Li, Malik, and Wolfe were very successful with an ILP-only approach to WCET determination [11,12,13,14] ... at least as far as getting papers about their approach published. Cache and pipeline behavior prediction are formulated as a single linear program. The i960KB is investigated, a 32-bit microprocessor with a 512 byte direct mapped instruction cache and a fairly simple pipeline. Only structural hazards need to be modeled, thus keeping the complexity of the integer linear program moderate compared to the expected complexity of a model for a modern microprocessor. Variable execution times, branch prediction, and instruction prefetching are not considered at all. Using this approach for superscalar pipelines does not seem very promising, considering the analysis times reported in the article.

One of the severe problems is the exponential increase of the size of the ILP in the number of competing l-blocks. *l-blocks* are maximally long contiguous sequences of instructions in a basic block mapped to the same cache set. Two l-blocks mapped to the same cache set *compete* if they do not have the same

address tag. For a fixed cache architecture, the number of competing l-blocks grows linearly with the size of the program. Differentiation by contexts, absolutely necessary to achieve precision, increases this number additionally. Thus, the size of the ILP is exponential in the size of the program. Even though the problem is claimed to be a network-flow problem the size of the ILP is killing the approach. Growing associativity of the cache increases the number of competing l-blocks. Thus, also increasing cache-architecture complexity plays against this approach.

Nonetheless, their method of modeling the control flow as an ILP, the so-called *Implicit Path Enumeration*, is elegant and can be efficient if the size of the ILP is kept small. It has been adopted by many groups working in this area.

The weakness of my argumentation against an ILP-only approach to WCET determination is apparent; I argue against one particular way of modeling the problem in one ILP. There could be other ways of modeling it.

6 Conclusion

The AI+ILP approach has proved successful in industrial practice. Results, at least on disciplined software, but quite complex processors are precise [24] and are produced quickly. Benchmark results reported there show an average overestimation of execution times of roughly 15%. This should be contrasted with results of an experiment by Alfred Rosskopf at EADS [19]. He measured a slow down of a factor of 30 when the caches of a PowerPC 603 were switched off. The benchmark described in [24] consisted of 12 tasks of roughly 13 kByte instructions each. The analysis of these tasks took less than 30 min. per task.

The approach is modular, in that it separates processor behaviour prediction and worst-case path determination. Reasons for the efficiency of the approach are twofold: AI used for the prediction of processor behaviour uses only one, albeit complex fixpoint computation, in which it verifies as many safety properties, i.e. absence of timing accidents, as it can. The remaining ILP modeling the control flow of the program, even when expanded to realize context differentiation, is still of moderate size and can be solved quickly.

As Ken McMillan pointed out, model checking may deliver more precise results since it exhaustively checks all possibilities, while abstract interpretation may loose information due to abstraction. Our practical experience so far does not show a significant loss of precision. On the other hand, none of the described alternatives of using MC seems to offer acceptable performance. It could turn out that the exponential worst case does not show up in practice as is the case in the experiments on byte-code verification reported in [2].

MC could check the conjunction of a set of safety properties in polynomial time, after these have been derived by abstract interpretation.

Only one ILP-only approach could be found in the literature. It had severe scaling problems as the size of the ILP grows exponentially with the size of the program to be analyzed.

Acknowledgements. Mooly Sagiv, Jens Palsberg, and Kim Guldstrand Larsen confronted me with the claim, that we could have used MC instead of AI + ILP.

This motivated me to think about why not. Andreas Podelski clarified my view of things while defending their claim. Alberto Sangiovanni-Vincentelli suggested that [11,13,12,14] had already solved the problem using ILP provoking me to find out, why they hadn't. Many thanks go to Reinhold Heckmann, Stephan Thesing, and Sebastian Winkel for valuable discussions and detailed hints and to the whole group in the Compiler Design Laboratory and at AbsInt for their cooperation on this successful line of research and development.

References

1. M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache Behavior Prediction by Abstract Interpretation. In *Proceedings of SAS'96, Static Analysis Symposium*, volume 1145 of *LNCS*, pages 52–66. Springer-Verlag, September 1996.
2. D. Basin, S. Friedrich, M. Gawkowski, and J. Posegga. Bytecode model checking: An experimental analysis. In *9th International SPIN Workshop on Model Checking of Software*, volume 2318 of *LNCS*, pages 42–59. Springer-Verlag, 2002.
3. P. Cousot. Automatic verification by abstract interpretation, invited talk. *International Symposium in Honor of Zohar Manna*, Taormina, Sicily, Italy, Tuesday, July 1st, 2003.
4. Andreas Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, 2003.
5. C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *EMSOFT*, volume 2211 of *LNCS*, pages 469 – 485, 2001.
6. C. Ferdinand, F. Martin, and R. Wilhelm. Cache Behavior Prediction by Abstract Interpretation. *Science of Computer Programming*, 35:163 – 189, 1999.
7. Christian Ferdinand. Cache Behavior Prediction for Real-Time Systems. PhD Thesis, Universität des Saarlandes, September 1997.
8. Christopher A. Healy, David B. Whalley, and Marion G. Harmon. Integrating the Timing Analysis of Pipelining and Instruction Caching. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 288–297, December 1995.
9. Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *IEEE Proceedings on Real-Time Systems*, 91(7):1038–1054, 2003.
10. M. Langenbach, St. Thesing, and R. Heckmann. Pipeline modelling for timing analysis. In *Static Analysis Symposium*, LNCS. Springer-Verlag, 2002.
11. Yau-Tsun Steven Li and Sharad Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, pages 456–461, June 1995.
12. Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 298–307, December 1995.
13. Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Performance Estimation of Embedded Software with Instruction Cache Modeling. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 380–387, November 1995.
14. Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1996.

15. F. Martin, M. Alt, R. Wilhelm, and C. Ferdinand. Analysis of Loops. In *Proceedings of the International Conference on Compiler Construction (CC'98)*, volume 1383 of *LNCS*, pages 80–94. Springer-Verlag, 1998.
16. Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
17. Chang Yun Park and Alan C. Shaw. Experiments with a Program Timing Tool Based on Source-Level Timing Schema. *IEEE Computer*, 24(5):48–57, May 1991.
18. P. Puschner and Ch. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Real-Time Systems*, 1:159–176, 1989.
19. Alfred Rosskopf. Vergleich der Performance von zwei Plattformen für Ada-Applikationen: PowerPC/ObjectAda gegenüber MC68020/XDAda. Talk at Ada Deutschland, March 2001.
20. J. Schneider and C. Ferdinand. Pipeline Behaviour Prediction for Superscalar Processors by Abstract Interpretation. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, Atlanta, June 1999.
21. H. Theiling and C. Ferdinand. Combining Abstract Interpretation and ILP for Microarchitecture Modelling and Program Path Analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 144–153, Madrid, Spain, December 1998.
22. H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2/3):157–179, May 2000.
23. Henrik Theiling. *Control Flow Graphs for Real-Time System Analysis*. PhD thesis, Universität des Saarlandes, 2002.
24. S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand. An abstract interpretation-based timing validation of hard real-time avionics software systems. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN 2003)*, pages 625–632. IEEE Computer Society, June 2003.
25. Stephan Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, Universität des Saarlandes. forthcoming.

A Grand Challenge for Computing: Towards Full Reactive Modeling of a Multi-cellular Animal*

David Harel

Faculty of Mathematics and Computer Science,
Weizmann Institute of Science,
Rehovot, 76100 Israel dharel@weizmann.ac.il

Abstract. Biological systems can be modeled beneficially as reactive systems, using languages and tools developed for the construction of man-made systems. Our long-term aim is to model a full multi-cellular animal as a reactive system; specifically, the C. elegans nematode worm, which is complex, but very well-defined in terms of anatomy and genetics. The challenge is to construct a full, true-to-all-known-facts, 4-dimensional, fully animated model of the development and behavior of this worm (or of a comparable multi-cellular animal), which is multi-level and interactive, and is easily extendable – even by biologists – as new biological facts are discovered.

The proposal has three premises: (i) that satisfactory frameworks now exist for reactive system modeling and design; (ii) that biological research is ready for an extremely significant transition from analysis (reducing experimental observations to elementary building blocks) to synthesis (integrating the parts into a comprehensive whole), a transition that requires mathematics and computation; and (iii) that the true complexity of the dynamics of biological systems – specifically multi-cellular living organisms – stems from their reactivity.

In earlier work on T-cell reactivity, we addressed the feasibility of modeling biological systems as reactive systems, and the results were very encouraging [1]. Since then, we have turned to two far more complex systems, with the intention of establishing the basis for addressing the admittedly extremely ambitious challenge outlined above. One is modeling T-cell behavior in the thymus [2], using statecharts and Rhapsody, and the other is on VPC fate acquisition in the egg-laying system of C. elegans [3], for which we used LSCs and the Play-Engine [4].

The proposed long term effort could possibly result in an unprecedented tool for the research community, both in biology and in computer science. We feel that much of the research in systems biology will be going this way in the future: grand efforts at using computerized system modeling and analysis techniques for understanding complex biology.

* Full paper in EATCS Bulletin, European Association for Theoretical Computer Science, October, 2003. (Early version prepared for the UK Workshop on Grand Challenges in Computing Research, November 2002.)

B. Steffen and G. Levi (Eds.): VMCAI 2004, LNCS 2937, pp. 323–324, 2004.

References

1. N. Kam, I.R. Cohen and D. Harel. The Immune System as a Reactive System: Modeling T Cell Activation with Statecharts. *Bull. Math. Bio., to appear. (Extended abstract in Proc. Visual Languages and Formal Methods (VLFM'01), part of IEEE Symp. on Human-Centric Computing (HCC'01)*, pp. 15–22, 2001.
2. S. Efroni, D. Harel and Irun. R. Cohen. Towards Rigorous Comprehension of Biological Complexity: Modeling, Execution and Visualization of Thymic T Cell Maturation. *Genome Research, 13 (2003),* 2485–2484., 2003.
3. N. Kam, D. Harel, H. Kugler, R. Marelly, A. Pnueli, E.J.A. Hubbard and M.J. Stern. Formal Modeling of C. elegans Development: A Scenario-Based Approach. *Proc. Int. Workshop on Computational Methods in Systems Biology (ICMSB 2003)*, February, 2003.
4. D. Harel and R. Marelly. Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. *Springer-Verlag, 2003.*

Author Index