Eran Yahav (Ed.)

ARCoSS

LNCS 6887

Static Analysis

18th International Symposium, SAS 2011
Venice, Italy, September 2011
Proceedings

Springer

Lecture Notes in Computer Science 6887

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Advanced Research in Computing and Software Science

Subline of Lectures Notes in Computer Science

Eran Yahav (Ed.)

Static Analysis

18th International Symposium, SAS 2011
Venice, Italy, September 14-16, 2011
Proceedings

Springer

Volume Editor

Eran Yahav
Technion
Computer Science Department
Technion City
Haifa 32000, Israel
E-mail: yahave@cs.technion.ac.il

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-23701-0 e-ISBN 978-3-642-23702-7
DOI 10.1007/978-3-642-23702-7
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011935065

CR Subject Classification (1998): D.2, F.3, D.3, D.2.4, F.4.1, D.1

LNCS Sublibrary: SL 2 – Programming and Software Engineering

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

Static analysis is increasingly recognized as a fundamental tool for program verification, bug detection, compiler optimization, program understanding, and software maintenance. The series of Static Analysis Symposia has served as the primary venue for presentation of theoretical, practical, and application advances in the area.

This volume contains the proceedings of the 18th International Static Analysis Symposium (SAS 2011), which was held September 14–16, 2011, in Venice, Italy. Previous symposia were held in Perpignan, France (2010); Los Angeles, USA (2009); Valencia, Spain (2008); Kongens Lyngby, Denmark (2007); Seoul, South Korea (2006); London, UK (2005); Verona, Italy (2004); San Diego, USA (2003); Madrid, Spain (2002); Paris, France (2001); Santa Barbara, USA (2000); Venice, Italy (1999); Pisa, Italy (1998); Paris, France (1997); Aachen, Germany (1996); Glasgow, UK (1995); and Namur, Belgium (1994).

The 18th International Static Analysis Symposium (SAS 2011), was held together with three affiliated Workshops: NSAD 2011 (The Third Workshop on Numerical and Symbolic Abstract Domains), SASB (Second Workshop on Static Analysis and Systems Biology) on September 13, 2011, and TAPAS 2010 (Tools for Automatic Program AnalysiS) on September 17, 2011.

There were 67 submissions. Each submission was reviewed by at least four Program Committee members. The committee decided to accept 22 papers.

In addition to the 22 contributed papers, the program included five invited talks by Jérôme Feret (Ecole Normale Superieure, France), Daniel Kaestner (AbsInt, Germany), Ken McMillan (Microsoft Research, USA), John Mitchell (Stanford, USA), and Sriram Rajamani (Microsoft Research, India).

On behalf of the Program Committee, I would like to thank all the external referees for their participation in the reviewing process. We are grateful to our generous sponsors, to all the members of the Organizing Committee in Venice for their hard work, and to the EasyChair team for the use of their system.

September 2011 Eran Yahav

Conference Organization

General Chairs

Gilberto Filè	University of Padova, Italy
Mooly Sagiv	Tel Aviv University, Israel

Program Chair

Eran Yahav	Technion, Israel

Program Committee

Anindya Banerjee	IMDEA Software Institute, Spain
Michele Bugliesi	Università Cà Foscari, Italy
Byron Cook	Microsoft Research, UK
Radhia Cousot	École Normale Supérieure and CNRS, France
Roberto Giacobazzi	University of Verona, Italy
Sumit Gulwani	Microsoft Research, USA
Chris Hankin	Imperial College London, UK
Naoki Kobayashi	Tohoku University, Japan
Viktor Kuncak	EPFL, Switzerland
Ondrej Lhotak	University of Waterloo, Canada
Matthieu Martel	Université de Perpignan, France
Antoine Miné	École Normale Supérieure and CNRS, France
George Necula	UC Berkeley, USA
Ganesan Ramalingam	Microsoft Research, India
Francesco Ranzato	University of Padova, Italy
Thomas Reps	University of Wisconsin, USA
Noam Rinetzky	Queen Mary University of London, UK
Helmut Seidl	University of Munich, Germany
Zhendong Su	University of California, Davis, USA
Hongseok Yang	University of Oxford, UK

Organizing Committee

Alberto Carraro	Università Cà Foscari
Nicoletta Cocco	Università Cà Foscari
Sabina Rossi	Università Cà Foscari, Chair
Silvia Crafa	University of Padova

Steering Committee

Patrick Cousot	École Normale Supérieure, France and New York University, USA
Radhia Cousot	École Normale Supérieure and CNRS, France
Roberto Giacobazzi	University of Verona, Italy
Gilberto Filè	University of Padova, Italy
Manuel Hermenegildo	IMDEA Software Institute, Spain
David Schmidt	Kansas State University, USA

External Reviewers

Reynald Affeldt
Xavier Allamigeon
Gianluca Amato
Saswat Anand
Tycho Andersen
Domagoj Babic
Earl Barr
Matko Botincan
Olivier Bouissou
Jacob Burnim
Stefano Calzavara
Bor-Yuh Evan Chang
Alexandre Chapoutot
Swarat Chaudhuri
Hana Chockler
Patrick Cousot
Silvia Crafa
Ferruccio Damiani
Eva Darulova
Giorgio Delzanno
Alessandra Di Pierro
Dino Distefano
Mike Dodds
Matt Elder
Tayfun Elmas
Michael Emmi
Karine Even
Jérôme Feret
Pietro Ferrara
Mark Gabel
Pierre Ganty
Pierre-Loic Garoche
Denis Gopan
Nikos Gorogiannis
Alexey Gotsman
Radu Grigore
Guy Gueta
Tihomir Gvero
Christoph Haase
Raju Halder
Emmanuel Haucourt
Hossein Hojjat
Kazuhiro Inaba
Arnault Ioualalen
Swen Jacobs
Pierre Jouvelot
Andy King
Etienne Kneuss
Eric Koskinen
Joerg Kreiker
Sava Krstic
Michael Kuperstein
Taeho Kwon
Ali Sinan Köksal
Vincent Laviron
Matt Lewis
David Lo
Giuseppe Maggiore
Stephen Magill
Roman Manevich
Mark Marron
Damien Massé
Isabella Mastroeni
Laurent Mauborgne
Michael Monerau
David Monniaux

Mayur Naik
Do Thi Bich Ngoc
Durica Nikolic
Pavel Parizek
Nimrod Partush
Rasmus Lerchedahl Petersen
Carla Piazza
Ruzica Piskac
Sebastian Pop
Prathmesh Prabhu
Mohammad Raza
Guillaume Revy
Xavier Rival
Philipp Ruemmer
Sriram Sankaranarayanan
Hamadou Sardaouna
Francesca Scozzari
Ohad Shacham
Tushar Sharma
Rishabh Singh
Harald Søndergaard
Fausto Spoto
Manu Sridharan
Saurabh Srivastava
Christos Stergiou
Kohei Suenaga
Philippe Suter
Aditya Thakur
Hiroshi Unno
Martin Vechev
Jules Villard
Dimitrios Vytiniotis
Björn Wachter
Thomas Wies
Herbert Wiklicky
Liang Xu
Greta Yorsh
Enea Zaffanella
Florian Zuleger
Cameron Zwarich

Table of Contents

Widening and Interpolation

Kenneth L. McMillan

Microsoft Research

Abstract. Widening/narrowing and interpolation are two techniques for deriving a generalization about unbounded behaviors from an analysis of bounded behaviors. The purpose of both methods is to produce an inductive invariant that proves some property of a program or other discrete dynamic system. In the case of widening, we obtain a guess at an inductive invariant by extrapolating a sequence of approximations of the program behavior, either forward or backward. In the case of interpolation, we use the intermediate assertions in proofs of correctness of bounded behaviors.

To contrast these approaches, we will view widening/narrowing operators as deduction systems that have been weakened in some way in order to force generalization. From this point of view, we observe some important similarities and differences between the methods. Both methods are seen to derive candidate inductive invariants from proofs about bounded execution sequences. In the case of widening/narrowing, we produce the strongest k-step post-condition (or weakest k-step pre-condition) derivable in the weakened proof system. By contrast, using interpolation, we derive candidate inductive invariants from a *simple* proof of safety a k-step sequence. The intermediate assertions we infer are neither the strongest nor the weakest possible, but are merely sufficient to prove correctness of the bounded sequence. In widening/narrowing there is an asymmetry in the treatment of the initial and final conditions, since we widen either forward or backward. In interpolation, there is no preferred direction. The initial and final conditions of the sequence are dual.

The most salient distinction between the two approaches is in their *inductive bias*. Any kind of generalization requires some *a priori* preference for one form of statement over another. In the case of widening, this bias is explicit, and given as an *a priori* weakening of the deduction system. In interpolation, we do not weaken the deduction system, but rather bias in favor of parsimonious proofs in the given system. This can be viewed as an application of Occam's razor: proofs using fewer concepts are more likely to generalize. The inductive bias in interpolation is thus less direct than in widening/narrowing as it derives from the chosen logic, and some notion of cost associated to proofs.

E. Yahav (Ed.): SAS 2011, LNCS 6887, p. 1, 2011.

Program Analysis and Machine Learning: A Win-Win Deal

Aditya V. Nori and Sriram K. Rajamani

Microsoft Research India
{adityan,sriram}@microsoft.com

We give an account of our experiences working at the intersection of two fields: program analysis and machine learning. In particular, we show that machine learning can be used to infer annotations for program analysis tools, and that program analysis techniques can be used to improve the efficiency of machine learning tools.

Every program analysis tool needs annotations. Type systems need users to specify types. Program verification tools need users to specify preconditions, postconditions and invariants in some form. Information flow analyzers require users to specify sources and sinks for taint, and sanitizers, which cleanse taint. We show how such annotations can be derived from high level intuitions using Bayesian inference. In this approach, annotations are thought of as random variables, and intuitions of the programmer are stated as probabilistic constraints over these random variables. The Bayesian framework models and tolerates uncertainty in programmer intuitions, and Bayesian inference is used to infer most likely annotations, given the program structure and programmer intuitions. We give specific examples of such annotation inference for information flow [5] and ownership types [1]. We also describe a generic scheme to infer annotations for any safety property.

Machine learning algorithms perform statistical inference by analyzing voluminous data. Program analysis techniques can be used to greatly optimize these algorithms. In particular, statistical inference tools [3,6] perform inference from data and first-order logic specifications. We show how Counterexample Guided Abstraction Refinement (CEGAR) techniques, commonly used in verification tools and theorem provers can be used to lazily instantiate axioms and improve the efficiency of inference [2]. This approach also enables users of these tools to express their models with rich theories such as linear arithmetic and uninterpreted functions. There is a recent trend in the machine learning community to specify machine learning models as programs [4]. Inspired by this view of models as programs, we show how program analysis techniques such as backward analysis and weakest preconditions can be used to improve the efficiency of algorithms for learning tasks such as the computation of posterior probabilities given some observed data.

In summary, we believe that these cross fertilization of ideas from program analysis and machine learning have the potential to improve both fields, resulting in a mutual win-win deal. We speculate on further opportunities for mutually beneficial exchange of ideas between the two fields.

E. Yahav (Ed.): SAS 2011, LNCS 6887, pp. 2–3, 2011.

References

1. Beckman, N., Nori, A.V.: Probabilistic, modular and scalable inference of typestate specifications. In: Programming Languages Design and Implementation (PLDI), pp. 211–221 (2011)
2. Chaganty, A., Lal, A., Nori, A.V., Rajamani, S.K.: Statistical inference modulo theories. Technical report, Microsoft Research (2011)
3. Kok, S., Sumner, M., Richardson, M., Singla, P., Poon, H., Lowd, D., Domingos, P.: The Alchemy system for statistical relational AI. Technical report, University of Washington, Seattle (2007), http://alchemy.cs.washington.edu
4. Koller, D., McAllester, D.A., Pfeffer, A.: Effective Bayesian inference for stochastic programs. In: Fifteenth National Conference on Artificial Intelligence (AAAI), pp. 740–747 (1997)
5. Livshits, V.B., Nori, A.V., Rajamani, S.K., Banerjee, A.: Merlin: Specification inference for explicit information flow problems. In: Programming Languages Design and Implementation (PLDI), pp. 75–86 (2009)
6. Niu, F., Re, C., Doan, A., Shavlik, J.: Tuffy: Scaling up statistical inference in Markov logic networks using an RDBMS. In: International Conference on Very Large Data Bases, VLDB (2011)

Program Analysis for Web Security

John C. Mitchell

Stanford University

Abstract. The evolving nature of web applications and the languages they are written in continually present new challenges and new research opportunities. For example, web sites that present trusted and untrusted code to web users aim to provide isolation and secure mediation across a defined interface. Older versions of JavaScript make it difficult for one section of code to provide limited access to another, while improvements in standardized ECMAScript bring the problem closer to traditional language-based encapsulation. As a result, rigorous language semantics and acceptable limitations on the language constructs used in trusted code make provable solutions possible.

We have developed sound program analysis tools for specific versions of ECMAScript 5, providing security guarantees against threats from untrusted code in a larger language. However, many security problems remain and there are many ways that future language tools may improve web security and developer productivity.

E. Yahav (Ed.): SAS 2011, LNCS 6887, p. 4, 2011.

Astrée: Design and Experience

Daniel Kästner

AbsInt Angewandte Informatik GmbH
Science Park 1, D-66123 Saarbrücken, Germany

Abstract. Safety-critical embedded software has to satisfy stringent quality requirements. Testing and validation consumes a large and growing fraction of development cost. One class of errors which are hard to find by testing are runtime errors, e.g., arithmetic overflows, array bound violations, or invalid pointer accesses. The consequences of runtime errors range from erroneous program behavior to crashes. Since they are non-functional errors, it is usually not possible to achieve reasonable coverage by writing a set of specific test cases.

Unsound static analysis tools can find some bugs, but there is no guarantee that all bugs have been detected. Sound static runtime error analyzers provide full control and data coverage so that every potential runtime error is discovered. When the analyzer reports zero alarms, the absence of runtime errors has been proven. However they can produce false alarms: any alarm which is not reported as a definite error might be a true error, or a false alarm. In the past, usually there were so many false alarms that manually inspecting each alarm was too time-consuming. Therefore not all alarms could be removed and no proof of the absence of runtime errors could be given.

Astrée is a sound static analyzer designed to find all potential runtime errors in C programs while achieving zero false alarms. It has successfully been used to analyze large-scale safety-critical avionics software with zero false alarms. This talk gives an overview of the history and the design of Astrée, discusses important industry requirements, and illustrates the industrialization process from an academical research tool to a commercial product. It also outlines ongoing development and future research issues.

E. Yahav (Ed.): SAS 2011, LNCS 6887, p. 5, 2011.

Formal Model Reduction

Jérôme Feret

Laboratoire d'informatique de l'École normale supérieure
(INRIA/ÉNS/CNRS)
www.di.ens.fr/~feret

Modelers of molecular signaling networks must cope with the combinatorial explosion of protein states generated by post-translational modifications and complex formations. Rule-based models provide a powerful alternative to approaches that require an explicit enumeration of all possible molecular species of a system [1,2]. Such models consist of formal rules stipulating the (partial) contexts for specific protein-protein interactions to occur. The behavior of the models can be formally described by stochastic or differential semantics. Yet, the naive computation of these semantics does not scale to large systems, because it does not exploit the lower resolution at which rules specify interactions.

We present a formal framework for constructing coarse-grained systems. We instantiate this framework with two abstract domains. The first one tracks the flow of information between the different regions of chemical species, so as to detect and abstract away some useless correlations between the state of sites of molecular species. The second one detects pairs of sites having the same capabilities of interactions, and abstract away any distinction between them.

The result of our abstraction is a set of molecular patterns, called fragments, and a system which describes exactly the concentration (or population) evolution of these fragments. The method never requires the execution of the concrete rule-based model and the soundness of the approach is described and proved by abstract interpretation [3]. Unlike our previous analysis [4], our fragments are heterogeneous. The cutting of a protein into portions may depend on its position within the molecular species. This matches more closely with the flow of information. Indeed, within a molecular species, the behavior of a protein may be driven by the state of a site without being driven by the state of the same site in other instances of the protein. Our new analysis exploits this efficiently.

(Joint work with F. Camporesi, V. Danos, W. Fontana, R. Harmer, and J. Krivine.)

References

1. Danos, V., Laneve, C.: Formal molecular biology. TCS 325(1) (2004)
2. Blinov, M.L., Faeder, J.R., Hlavacek, W.S.: BioNetGen: software for rule-based modeling of signal transduction based on the interactions of molecular domains. Bioinformatics 20 (2004)
3. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL (1977)
4. Feret, J., Danos, V., Krivine, J., Harmer, R., Fontana, W.: Internal coarse-graining of molecular systems. PNAS (2009)

E. Yahav (Ed.): SAS 2011, LNCS 6887, p. 6, 2011.

Purity Analysis: An Abstract Interpretation Formulation

Ravichandhran Madhavan, Ganesan Ramalingam, and Kapil Vaswani

Microsoft Research, India
{t-rakand,grama,kapilv}@microsoft.com

Abstract. Salcianu and Rinard present a compositional purity analysis that computes a summary for every procedure describing its side-effects. In this paper, we formalize a generalization of this analysis as an abstract interpretation, present several optimizations and an empirical evaluation showing the value of these optimizations. The Salcianu-Rinard analysis makes use of abstract heap graphs, similar to various heap analyses and computes a shape graph at every program point of an analyzed procedure. The key to our formalization is to view the shape graphs of the analysis as an *abstract state transformer* rather than as a set of abstract states: the concretization of a shape graph is a function that maps a concrete state to a set of concrete states. The abstract interpretation formulation leads to a better understanding of the algorithm. More importantly, it makes it easier to change and extend the basic algorithm, while guaranteeing correctness, as illustrated by our optimizations.

1 Introduction

Compositional or modular analysis [6] is a key technique for scaling static analysis to large programs. Our interest is in techniques that analyze a procedure in isolation, using pre-computed summaries for called procedures, computing a summary for the analyzed procedure. Such analyses are widely used and have been found to scale well. In this paper we consider an analysis presented by Salcianu and Rinard [17], based on a pointer analysis due to Whaley and Rinard [19], which we will refer to the WSR analysis. Though referred to as a purity analysis, it is a more general-purpose analysis that computes a summary for every procedure, in the presence of dynamic memory allocation, describing its side-effects. This is one of the few heap analyses that is capable of treating procedures in a compositional fashion.

WSR analysis is interesting for several reasons. Salcianu and Rinard present an application of the analysis to classify a procedure as *pure* or *impure*, where a procedure is impure if its execution can potentially modify pre-existing state. Increasingly, new language constructs (such as iterators, parallel looping constructs and SQL-like query operators) are realized as higher-order library procedures with procedural parameters that are expected to be side-effect free. Purity checkers can serve as verification/bug-finding tools to check usage of these constructs. Our interest in this analysis stems from our use of an extension

E. Yahav (Ed.): SAS 2011, LNCS 6887, pp. 7–24, 2011.

of this analysis to statically verify the correctness of the use of speculative parallelism [13]. WSR analysis can also help more sophisticated verification tools, such as [8], which use simpler analyses to identify procedure calls that do not affect properties of interest to the verifier and can be abstracted away.

However, we felt the need for various extensions of the WSR analysis. A key motivation was efficiency. Real-world applications make use of large libraries such as the base class libraries in .NET. While the WSR analysis is reasonably efficient, we find that it still does not scale to such libraries. Another motivation is increased functionality: our checker for speculative parallelism [13] needs some extra information (must-write sets) beyond that computed by the analysis. A final motivating factor is better precision: the WSR analysis declares "pure" procedures that use idioms like lazy initialization and caching as impure.

The desire for these extensions leads us to formulate, in this paper, the WSR analysis as an abstract interpretation, to simplify reasoning about the soundness of these extensions. The formulation of the WSR analysis as an abstract interpretation is, in fact, mentioned as an open problem by Salcianu ([16], page 128).

The WSR analysis makes use of abstract heap graphs, similar to various heap analyses and computes a shape graph g_u at every program point u of an analyzed procedure. The key to our abstract interpretation formulation, however, is to view a shape graph utilized by the analysis as an *abstract state transformer* rather than as a set of abstract states: thus, the concretization of a shape graph is a function that maps a concrete state to a set of concrete states. Specifically, if the graph computed at program point u is g_u, then for any concrete state σ, $\gamma(g_u)(\sigma)$ conservatively approximates the set of states that can arise at program point u in the execution of the procedure on an initial state σ. In our formalization, we present a concrete semantics in the style of the functional approach to interprocedural analysis presented by Sharir and Pnueli. The WSR analysis can then be seen as a natural abstract interpretation of this concrete semantics.

We then present three optimizations viz. duplicate node merging, summary merging, and safe node elimination, that improve the efficiency of WSR analysis. We use the abstract interpretation formulation to show that these optimizations are sound. Our experiments show that these optimizations significantly reduce both analysis time (sometimes by two orders of magnitude or more) and memory consumption, allowing the analysis to scale to large programs.

2 The Language, Concrete Semantics, and the Problem

Syntax. A program consists of a set of procedures. A procedure P consists of a control-flow graph, with an entry vertex $entry(P)$ and an exit vertex $exit(P)$. The entry vertex has no predecessor and the exit vertex has no successor. Every edge of the control-flow graph is labelled by a primitive statement. The set of primitive statements are shown in Fig. 1. We use $u \xrightarrow{S} v$ to indicate an edge in the control-flow graph from vertex u to vertex v labelled by statement S.

Statement S	**Concrete semantics** $[\![S]\!]_c(\mathsf{V},\mathsf{E},\sigma)$
$v_1 = v_2$	$\{(\mathsf{V},\mathsf{E},\sigma[v_1 \mapsto \sigma(v_2)]\}$
$v = \mathit{new}\ C$	$\{(\mathsf{V}\cup\{n\},\mathsf{E}\cup\{n\}\times \mathit{Fields}\times\{\mathit{null}\},\sigma[v\mapsto n]) \mid n\in N_c\setminus \mathsf{V}\}$
$v_1.f = v_2$	$\{(\mathsf{V},\{\langle u,l,v\rangle\in\mathsf{E} \mid u\neq\sigma(v_1)\vee l\neq f\}\cup\{\langle\sigma(v_1),f,\sigma(v_2)\rangle\},\sigma)\}$
$v_1 = v_2.f$	$\{(\mathsf{V},\mathsf{E},\sigma[v_1\mapsto n]) \mid \langle\sigma(v_2),f,n\rangle\in\mathsf{E}\}$
$\mathit{Call}\ P(v_1,\cdots,v_k)$	Semantics defined below

Fig. 1. Primitive statements and their concrete semantics

Concrete Semantics Domain. Let *Vars* denote the set of variable names used in the program, partitioned into the following disjoint sets: the set of global variables *Globals*, the set of local variables *Locals* (assumed to be the same for every procedure), and the set of formal parameter variables *Params* (assumed to be the same for every procedure). Let *Fields* denote the set of field names used in the program. We use a simple language in which all variables and fields are of pointer type. We use a fairly common representation of the concrete state as a concrete (points-to or shape) graph.

Let N_c be an unbounded set of locations used for dynamically allocated objects. A concrete state or points-to graph $g \in \mathbb{G}_c$ is a triple $(\mathsf{V},\mathsf{E},\sigma)$, where $\mathsf{V} \subseteq N_c$ represents the set of objects in the heap, $\mathsf{E} \subseteq \mathsf{V} \times \mathit{Fields} \times \mathsf{V}$ (a set of labelled edges) represents values of pointer fields in heap objects, and $\sigma \in \Sigma_c = \mathit{Vars} \mapsto \mathsf{V}$ represents the values of program variables. In particular, $(u,f,v) \in \mathsf{E}$ iff the f field of the object u points to object v. We assume N_c includes a special element *null*. Variables and fields of new objects are initialized to *null*.

Let $\mathcal{F}_c = \mathbb{G}_c \mapsto 2^{\mathbb{G}_c}$ be the set of functions that map a concrete state to a set of concrete states. We define a partial order $\sqsubseteq_c$ on $\mathcal{F}_c$ as follows: $f_a \sqsubseteq_c f_b$ iff $\forall g \in \mathbb{G}_c.f_a(g) \subseteq f_b(g)$. Let $\sqcup_c$ denote the corresponding least upper bound (join) operation defined by: $f_a \sqcup_c f_b = \lambda g.f_a(g) \cup f_b(g)$. For any $f \in \mathcal{F}_c$, we define $\overline{f} : 2^{\mathbb{G}_c} \mapsto 2^{\mathbb{G}_c}$ by: $\overline{f}(G) = \cup_{g\in G} f(g)$. We define the "composition" of two functions in $\mathcal{F}_c$ as follows: $f_a \circ f_b = \lambda g.\overline{f_b}(f_a(g))$.

Concrete Semantics. Every primitive statement S has a semantics $[\![S]\!]_c \in \mathcal{F}_c$, as shown in Fig. 1. Every primitive statement has a label ℓ which is not used in the concrete semantics and is, hence, omitted from the figure. The execution of most statements transforms a concrete state to another concrete state, but the signature allows us to model non-determinism (e.g., dynamic memory allocation can return any unallocated object). The signature also allows us to model execution errors such as null-pointer dereference, though the semantics presented simplifies error handling by treating *null* as just a special object.

We now define a concrete summary semantics $[\![P]\!]_c \in \mathcal{F}_c$ for every procedure P. The semantic function $[\![P]\!]_c$ maps every concrete state g_c to the set of concrete states that the execution of P with initial state g_c can produce.

We introduce a new variable φ_u for every vertex in the control-flow graph (of any procedure) and a new variable $\varphi_{u,v}$ for every edge $u \to v$ in the control-flow graph. The semantics is defined as the least fixed point of the following set of equations. The value of φ_u in the least fixed point is a function that maps any concrete state g to the set of concrete states that arise at program point u when the procedure containing u is executed with an initial state g. Similarly, $\varphi_{u,v}$ captures the states after the execution of the statement labelling edge $u \rightarrow v$.

$$\varphi_v = \lambda g.\{g\} \qquad v \text{ is an entry vertex} \tag{1}$$
$$\varphi_v = \bigsqcup_c \{\varphi_{u,v} \mid u \rightarrow v\} \qquad v \text{ is not an entry vertex} \tag{2}$$
$$\varphi_{u,v} = \varphi_u \circ [\![S]\!]_c \qquad \text{where } u \xrightarrow{S} v \text{ and S is not a call-stmt} \tag{3}$$
$$\varphi_{u,v} = \varphi_u \circ \mathit{CallReturn}_S(\varphi_{exit(Q)}) \qquad \text{where } u \xrightarrow{S} v,\ \text{S is a call to proc Q} \tag{4}$$

The first three equations are straightforward. Consider Eq. 4, corresponding to a call to a procedure Q. The value of $\varphi_{exit(Q)}$ summarizes the effect of the execution of the whole procedure Q. In the absence of local variables and parameters, we can define the right-hand-side of the equation to be simply $\varphi_u \circ \varphi_{exit(Q)}$.

The function $\mathit{CallReturn}_S(f)$, defined below, first initializes values of all local variables (to *null*) and formal parameters (to the values of corresponding actual parameters), using an auxiliary function push_S. It then applies f, capturing the procedure call's effect. Finally, the original values of local variables and parameters (of the calling procedure) are restored from the state preceding the call, using a function pop_S. For simplicity, we omit return values from our language.

Let $\mathit{Param}(i)$ denote the i-the formal parameter. Let S be a procedure call statement "`Call Q(`a_1`,...,`a_k`)`". We define the functions $\mathit{push}_S \in \Sigma_c \mapsto \Sigma_c$, $\mathit{pop}_S \in \Sigma_c \times \Sigma_c \mapsto \Sigma_c$, and $\mathit{CallReturn}_S$ as follows:

$$\begin{aligned}
\mathit{push}_S(\sigma) &= \lambda v.\ v \in \mathit{Globals} \rightarrow \sigma(v) \mid v \in \mathit{Locals} \rightarrow \mathit{null} \mid v = \mathit{Param}(i) \rightarrow \sigma(a_i)\\
\mathit{pop}_S(\sigma, \sigma') &= \lambda v.\ v \in \mathit{Globals} \rightarrow \sigma'(v) \mid v \in \mathit{Locals} \cup \mathit{Params} \rightarrow \sigma(v)\\
\mathit{CallReturn}_S(f) &= \lambda(\mathsf{V}, \mathsf{E}, \sigma).\{(\mathsf{V}', \mathsf{E}', \mathit{pop}_S(\sigma, \sigma')) \mid (\mathsf{V}', \mathsf{E}', \sigma') \in f(\mathsf{V}, \mathsf{E}, \mathit{push}_S(\sigma))\}
\end{aligned}$$

We define $[\![P]\!]_c$ to be the value of $\varphi_{exit(P)}$ in the least fixed point of equations (1)-(4), which exists by Tarski's fixed point theorem. Specifically, let VE denote the set of vertices and edges in the given program. The above equations can be expressed as a single equation $\varphi = F^\natural(\varphi)$, where $F^\natural$ is a monotonic function from the complete lattice $\mathit{VE} \mapsto \mathcal{F}_c$ to itself. Hence, $F^\natural$ has a least fixed point.

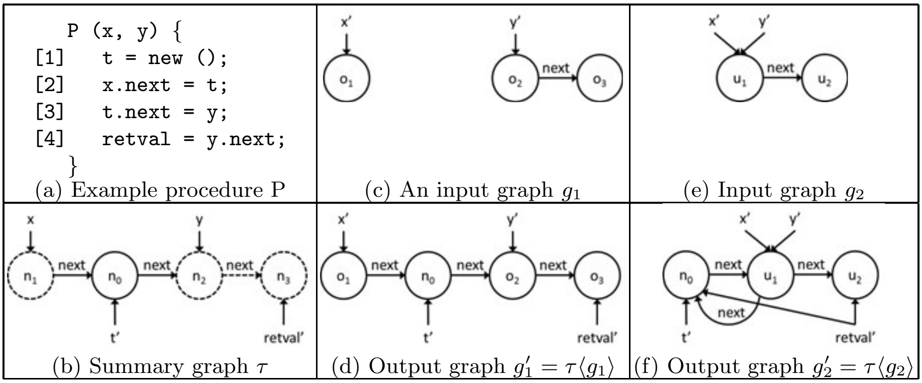
We note that the above collection of equations is similar to those used in Sharir and Pnueli's functional approach to interprocedural analysis [18] (extended by Knoop and Steffen [10]), with the difference that we are defining a concrete semantics here, while [18] is focused on abstract analyses. The equations are a simple functional version of the standard equations for defining a collecting semantics, with the difference that we are simultaneously computing a collecting semantics for every possible initial states of the procedure's execution.

The goal of the analysis is to compute an approximation of the set of quantities $[\![P]\!]_c$ using abstract interpretation.

3 The WSR Analysis as an Abstract Interpretation

3.1 Transformer Graphs: An Informal Overview

The WSR analysis uses a single abstract graph to represent a set of concrete states, similar to several shape and pointer analyses. The distinguishing aspect of the WSR analysis, however, is its extension of the graph based representation to represent (abstractions of) elements belonging to the functional domain $\mathcal{F}_c$. We now illustrate, using an example, how the graph representation is extended to represent an element of $\mathcal{F}_c = \mathbb{G}_c \mapsto 2^{\mathbb{G}_c}$. Consider the example procedure P shown in Fig. 2(a).



```
   P (x, y) {
[1]    t = new ();
[2]    x.next = t;
[3]    t.next = y;
[4]    retval = y.next;
   }
```

(a) Example procedure P

(c) An input graph g_1

(e) Input graph g_2

(b) Summary graph τ

(d) Output graph $g_1' = \tau\langle g_1 \rangle$

(f) Output graph $g_2' = \tau\langle g_2 \rangle$

Fig. 2. Illustration of transformer graphs

The summary graph τ computed for this procedure is shown in Fig. 2(b). (We omit the *null* node from the figures to keep them simple.) Vertices in a summary graph are of two types: *internal* (shown as circles with a solid outline) and *external* nodes (shown as circles with a dashed outline). Internal nodes represent new heap objects created during the execution of the procedure. E.g., vertex n_0 is an internal node and represents the object allocated in line 1. External nodes, in many cases, represent objects that exist in the heap when the procedure is invoked. In our example, n_1, n_2, and n_3 are external nodes.

Edges in the graph are also classified into *internal* and *external* edges, shown as solid and dashed edges respectively. The edges $n_1 \rightarrow n_0$ and $n_0 \rightarrow n_2$ are internal edges. They represent updates performed by the procedure (i.e., new points-to edges added by the procedure's execution) in lines 2 and 3. Edge $n_2 \rightarrow n_3$ is an external edge created by the dereference "`y.next`" in line 4. This edge helps identify the node(s) that the external node n_3 represents: namely, the objects obtained by dereferencing the `next` field of objects represented by n_2.

The summary graph τ indicates how the execution of procedure P transforms an initial concrete state. Specifically, consider an invocation of procedure P in an initial state given by graph g_1 shown in Fig. 2(c). The summary graph helps construct a transformed graph $g_1' = \tau\langle g_1\rangle$, corresponding to the state after the procedure's execution (shown in Fig. 2(d)) by identifying a set of new nodes and edges that must be added to g_1. (The underlying analysis performs no strong updates on the heap and, hence, never removes nodes or edges from the graph). We add a new vertex to g_1 for every internal node n in the summary graph. Every external node n in the summary graph represents a set of vertices $\eta(n)$ in g_1'. (We will explain later how the function η is determined by τ.) Every internal edge $u \xrightarrow{h} v$ in the summary graph identifies a set of edges $\{u' \xrightarrow{h} v' \mid u' \in \eta(u), v' \in \eta(v)\}$ that must be added to the graph g_1'. In our example, n_1, n_2 and n_3 represent, respectively, $\{o_1\}$, $\{o_2\}$ and $\{o_3\}$. This produces the graph shown in Fig. 2(d), which is an *abstract* graph representing a set of concrete states. The primed variables in the summary graph represent the (final) values of variables, and are used to determine the values of variables in the output graph.

An important aspect of the summary computed by the WSR analysis is that it can be used even in the presence of potential aliases in the input (or cutpoints [14]). Consider the input state g_2 shown in Fig. 2(e), in which parameters `x` and `y` point to the same object u_1. Our earlier description of how to construct the output graph still applies in this context. The main tricky aspect here is in correctly dealing with aliasing in the input. In the concrete execution, the update to `x.next` in line 2 updates the `next` field of object u_1. The aliasing between `x` and `y` means that `y.next` will evaluate to n_0 in line 4. Thus, in the concrete execution `retval` will point to the newly created object n_0 at the end of procedure execution, rather than u_2. This complication is dealt with in the definition of the mapping function η. For the example input g_2, the external node n_3 of the summary graph represents the set of nodes $\{u_2, n_0\}$. (This is an imprecise, but sound, treatment of the aliasing situation.) The rest of the construction applies just as before. This yields the abstract graph shown in Fig. 2(f).

More generally, an external node in the summary graph acts as a proxy for a set of *vertices in the final output graph to be constructed*, which may include nodes that exist in the input graph as well as new nodes added to the input graph (which themselves correspond to internal nodes of the summary graph).

We now define the transformer graph domain formally.

3.2 The Abstract Domain

The Abstract Graph Domain. We utilize a fairly standard abstract shape (or points-to) graph to represent a set of concrete states. Our formulation is parameterized by a given set N_a, the universal set of all abstract graph nodes. An abstract shape graph $g \in \mathbb{G}_a$ is a triple $(\mathsf{V}, \mathsf{E}, \sigma)$, where $\mathsf{V} \subseteq N_a$ represents the set of abstract heap objects, $\mathsf{E} \subseteq \mathsf{V} \times \mathit{Fields} \times \mathsf{V}$ (a set of labelled edges) represents possible values of pointer fields in the abstract heap objects, and $\sigma \in \mathit{Vars} \mapsto 2^{\mathsf{V}}$ is a map representing the possible values of program variables.

Given a concrete graph $g_1 = \langle \mathsf{V}_1, \mathsf{E}_1, \sigma_1\rangle$ and an abstract graph $g_2 = \langle \mathsf{V}_2, \mathsf{E}_2, \sigma_2\rangle$ we say that g_1 can be embedded into g_2, denoted $g_1 \preceq g_2$, if there exists a function $h : \mathsf{V}_1 \mapsto \mathsf{V}_2$ such that $\langle x, f, y\rangle \in \mathsf{E}_1 \Rightarrow \langle h(x), f, h(y)\rangle \in \mathsf{E}_2$ and

$\forall v \in \mathit{Vars}.\ \sigma_2(v) \supseteq \{h(\sigma_1(v))\}$. The concretization $\gamma_G(g_a)$ of an abstract graph g_a is defined to be the set of all concrete graphs that can be embedded into g_a:

$$\gamma_G(g_a) = \{g_c \in \mathbb{G}_c \mid g_c \preceq g_a\}$$

The Abstract Functional Domain. We now define the domain of graphs used to represent summary functions. A *transformer graph* $\tau \in \mathcal{F}_a$ is a tuple $(\mathsf{EV}, \mathsf{EE}, \pi, \mathsf{IV}, \mathsf{IE}, \sigma)$, where $\mathsf{EV} \subseteq N_a$ is the set of external vertices, $\mathsf{IV} \subseteq N_a$ is the set of internal vertices, $\mathsf{EE} \subseteq V \times \mathit{Fields} \times V$ is the set of external edges, where $V = \mathsf{EV} \cup \mathsf{IV}$, $\mathsf{IE} \subseteq V \times \mathit{Fields} \times V$ is the set of internal edges, $\pi \in (\mathit{Params} \cup \mathit{Globals}) \mapsto 2^V$ is a map representing the values of parameters and global variables in the *initial* state, and $\sigma \in \mathit{Vars} \mapsto 2^V$ is a map representing the possible values of program variables in the *transformed* state. Furthermore, a transformer graph τ is required to satisfy the following constraints:

$$\begin{aligned}\langle x, f, y\rangle \in \mathsf{EE} &\implies \exists u \in \mathit{range}(\pi).\text{x is reachable from u via } (\mathsf{IE} \cup \mathsf{EE}) \text{ edges}\\ y \in \mathsf{EV} &\implies y \in \mathit{range}(\pi) \vee \exists \langle x, f, y\rangle \in \mathsf{EE}\end{aligned}$$

Given a transformer graph $\tau = (\mathsf{EV}, \mathsf{EE}, \pi, \mathsf{IV}, \mathsf{IE}, \sigma)$, a node u is said to be a parameter node if $u \in \mathit{range}(\pi)$. A node u is said to be an escaping node if it is reachable from some parameter node via a path of zero or more edges (either internal or external). Let $\mathit{Escaping}(\tau)$ denote the set of escaping nodes in τ.

We now define the concretization function $\gamma_T : \mathcal{F}_a \rightarrow \mathcal{F}_c$. Given a transformer graph $\tau = (\mathsf{EV}, \mathsf{EE}, \pi, \mathsf{IV}, \mathsf{IE}, \sigma)$ and a concrete graph $g_c = (\mathsf{V}_c, \mathsf{E}_c, \sigma_c)$, we need to construct a graph representing the transformation of g_c by τ. As explained earlier, every external node $n \in \mathsf{EV}$ in the transformer graph represents a set of vertices in the transformed graph. We now define a function $\eta : (\mathsf{IV} \cup \mathsf{EV}) \mapsto 2^{(\mathsf{IV} \cup \mathsf{V}_c)}$ that maps each node in the transformer graph to a set of concrete nodes (in g_c) as well as internal nodes (in τ) as the least solution to the following set of constraints over variable μ.

$$v \in \mathsf{IV} \Rightarrow v \in \mu(v) \tag{5}$$

$$v \in \pi(\mathtt{X}) \Rightarrow \sigma_c(\mathtt{X}) \in \mu(v) \tag{6}$$

$$\langle u, f, v\rangle \in \mathsf{EE}, u' \in \mu(u), \langle u', f, v'\rangle \in \mathsf{E}_c \Rightarrow v' \in \mu(v) \tag{7}$$

$$\langle u, f, v\rangle \in \mathsf{EE}, \mu(u) \cap \mu(u') \neq \emptyset, \langle u', f, v'\rangle \in \mathsf{IE} \Rightarrow \mu(v') \subseteq \mu(v) \tag{8}$$

Explanation of the constraints: An internal node represents itself (Eq. 5). An external node labelled by a parameter `X` represents the node pointed to by `X` in the input state g_c (Eq. 6). An external edge $\langle u, f, v\rangle$ indicates that v represents any f-successor v' of any node u' represented by u in the input state (Eq. 7). However, with an external edge $\langle u, f, v\rangle$, we must also account for updates to the f field of the objects represented by u during the procedure execution, ie, the transformation represented by τ, via aliases (as illustrated by the example in Fig. 2(e)). Eq. 8 handles this. The precondition identifies u' as a potential alias for u (for the given input graph), and identifies updates performed on the f field of (nodes represented by) u'.

Given mapping function η, we define the transformed abstract graph $\tau\langle g_c\rangle$ as $\langle \mathsf{V}', \mathsf{E}', \sigma'\rangle$, where $\mathsf{V}' = \mathsf{V}_c \cup \mathsf{IV}$, $\mathsf{E}' = \mathsf{E}_c \cup \{\langle v_1, f, v_2\rangle \mid \langle u, f, v\rangle \in \mathsf{IE}, v_1 \in \eta(u), v_2 \in \eta(v)\}$ and $\sigma' = \lambda x. \bigcup_{u\in\sigma(x)} \eta(u)$. The transformed graph is an *abstract* graph that represents all concrete graphs that can be embedded in the abstract graph. Thus, we define the concretization function as below:

$$\gamma_T(\tau_a) = \lambda g_c.\gamma_G(\tau_a\langle g_c\rangle).$$

Our abstract interpretation formulation uses only a concretization function. There is no abstraction function α_T. While this form is less common, it is sufficient to establish the soundness of the analysis, as explained in [5]. Specifically, a concrete value $f \in \mathcal{F}_c$ is *correctly represented* by an abstract value $\tau \in \mathcal{F}_a$, denoted $f \sim \tau$, iff $f \sqsubseteq_c \gamma_T(\tau)$. We seek to compute an abstract value that correctly represents the least fixed point of the concrete semantic equations.

Containment Ordering. A natural "precision ordering" exists on $\mathcal{F}_a$, where τ_1 is said to be more precise than τ_2 iff $\gamma_T(\tau_1) \sqsubseteq_c \gamma_T(\tau_2)$. However, this ordering is not of immediate interest to us. (It is not even a partial order, and is hard to work with computationally.) We utilize a stricter ordering in our abstract fixed point computation. We define a relation $\sqsubseteq_{co}$ on $\mathcal{F}_a$ by: $(\mathsf{EV}_1, \mathsf{EE}_1, \pi_1, \mathsf{IV}_1, \mathsf{IE}_1, \sigma_1) \sqsubseteq_{co} (\mathsf{EV}_2, \mathsf{EE}_2, \pi_2, \mathsf{IV}_2, \mathsf{IE}_2, \sigma_2)$ iff $\mathsf{EV}_1 \subseteq \mathsf{EV}_2$, $\mathsf{EE}_1 \subseteq \mathsf{EE}_2$, $\forall x.\pi_1(x) \subseteq \pi_2(x)$, $\mathsf{IV}_1 \subseteq \mathsf{IV}_2$, $\mathsf{IE}_1 \subseteq \mathsf{IE}_2$, and $\forall x.\sigma_1(x) \subseteq \sigma_2(x)$.

Lemma 1. *$\sqsubseteq_{co}$ is a partial-order on $\mathcal{F}_a$ with a join operation, denoted $\sqcup_{co}$. Further, γ_T is monotonic with respect to $\sqsubseteq_{co}$: $\tau_1 \sqsubseteq_{co} \tau_2 \Rightarrow \gamma_T(\tau_1) \sqsubseteq_c \gamma_T(\tau_2)$.*

3.3 The Abstract Semantics

Our goal is to approximate the least fixed point computation of the concrete semantics equations 1-4. We do this by utilizing an analogous set of abstract semantics equations shown below. First, we fix the set N_a of abstract nodes. Recall that the domain $\mathcal{F}_a$ defined earlier is parameterized by this set. The WSR algorithm relies on an "allocation site" based merging strategy for bounding the size of the transformer graphs. We utilize the labels attached to statements as allocation-site identifiers. Let *Labels* denote the set of statement labels in the given program. We define N_a to be $\{n_x \mid x \in \mathit{Labels} \cup \mathit{Params} \cup \mathit{Globals}\}$.

We first introduce a variable ϑ_u for every vertex u in the control-flow graph (denoting the abstract value at a program point u), and a variable $\vartheta_{u,v}$ for every edge $u \to v$ in the control-flow graph (denoting the abstract value after the execution of the statement in edge $u \to v$).

$$\vartheta_v = \mathsf{ID} \qquad v \text{ is an entry vertex} \tag{9}$$

$$\vartheta_v = \sqcup_{co}\{\vartheta_{u,v} \mid u \xrightarrow{S} v\} \qquad v \text{ is not an entry vertex} \tag{10}$$

$$\vartheta_{u,v} = [\![S]\!]_a(\vartheta_u) \qquad \text{where } u \xrightarrow{S} v, \text{S is not a call-stmt} \tag{11}$$

$$\vartheta_{u,v} = \vartheta_{exit(Q)}\langle\!\langle\vartheta_u\rangle\!\rangle_a^S \qquad \text{where } u \xrightarrow{S} v, \text{S is a call to Q} \tag{12}$$

Statement S	**Abstract semantics** $[\![S]\!]_a \tau$ where $\tau = (\mathsf{EV}, \mathsf{EE}, \pi, \mathsf{IV}, \mathsf{IE}, \sigma)$
$v_1 = v_2$	$(\mathsf{EV}, \mathsf{EE}, \pi, \mathsf{IV}, \mathsf{IE}, \sigma[v_1 \mapsto \sigma(v_2)])$
$\ell : v = \mathit{new}\ C$	$(\mathsf{EV}, \mathsf{EE}, \pi, \mathsf{IV} \cup \{n_\ell\}, \mathsf{IE} \cup \{n_\ell\} \times \mathit{Fields} \times \{\mathit{null}\}, \sigma[v \mapsto \{n_\ell\}])$
$v_1.f = v_2$	$(\mathsf{EV}, \mathsf{EE}, \pi, \mathsf{IV}, \mathsf{IE} \cup \sigma(v_1) \times \{f\} \times \sigma(v_2), \sigma)$
$\ell : v_1 = v_2.f$	$\mathit{let}\ A = \{n \mid \exists n_1 \in \sigma(v_2), \langle n_1, f, n\rangle \in \mathsf{IE}\}\ \mathit{in}$ $\mathit{let}\ B = \sigma(v_2) \cap \mathit{Escaping}(\tau)\ \mathit{in}$ $\mathit{if}\ (B = \emptyset)$ $\mathit{then}\ (\mathsf{EV}, \mathsf{EE}, \pi, \mathsf{IV}, \mathsf{IE}, \sigma[v_1 \mapsto A])$ $\mathit{else}\ (\mathsf{EV} \cup \{n_\ell\}, \mathsf{EE} \cup B \times \{f\} \times \{n_\ell\}, \pi, \mathsf{IV}, \mathsf{IE}, \sigma[v \mapsto A \cup \{n_\ell\}])$

Fig. 3. Abstract semantics of primitive instructions

Here, ID is a transformer graph consisting of a external vertex for each global variable and each parameter (representing the identity function). Formally, $\mathsf{ID} = (\mathsf{EV}, \emptyset, \pi, \emptyset, \emptyset, \pi)$, where $\mathsf{EV} = \{n_x \mid x \in \mathit{Params} \cup \mathit{Globals}\}$ and $\pi = \lambda v.\ v \in \mathit{Params} \cup \mathit{Globals} \rightarrow n_v \mid v \in \mathit{Locals} \rightarrow \mathit{null}$. The abstract semantics $[\![S]\!]_a$ of any primitive statement S, other than a procedure call, is shown in Figure 3. The abstract semantics of a procedure call is captured by an operator $\tau_1 \langle\!\langle \tau_2 \rangle\!\rangle_a^S$, which we will define soon.

The abstract semantics of the first three statements are straightforward. The treatment of the dereference $v_2.f$ in the last statement is more involved. Here, the simpler case is where the dereferenced object is a non-escaping object: in this case, we can directly determine the possible values of $v_2.f$ from the information computed by the local analysis of the procedure. This is handled by the true branch of the conditional statement. The case of escaping objects is handled by the false branch. In this case, in addition to the possible values of $v_2.f$ identified by the local analysis, we must account for two sources of values unknown to the local analysis. The first possibility is that the dereferenced object is a pre-existing object (in the input state) with a pre-existing value for the f field. The second possibility is that the dereferenced object may have aliases unknown to the local analysis via which its f field may have been updated during the procedure's execution. We create an appropriate external node (with a corresponding incoming external edge) that serves as a proxy for these unknown values.

We now consider the abstract semantics of a procedure call statement. Let $\tau_r = (\mathsf{EV}_r, \mathsf{EE}_r, \pi_r, \mathsf{IV}_r, \mathsf{IE}_r, \sigma_r)$ be the transformer graph in the caller before a call statement S to Q and let $\tau_e = (\mathsf{EV}_e, \mathsf{EE}_e, \pi_e, \mathsf{IV}_e, \mathsf{IE}_e, \sigma_e)$ be the abstract summary of Q. We now show how to construct the graph $\tau_e \langle\!\langle \tau_r \rangle\!\rangle_a^S$ representing the abstract graph at the point after the method call. This operation is an extension of the operation $\tau\langle g_c \rangle$ used earlier to show how τ transforms a concrete state g_c into one of several concrete states.

We first utilize an auxiliary transformer $\tau_e \langle\!\langle \tau_r, \eta \rangle\!\rangle$ that takes an extra parameter η that maps nodes of τ_e to a set of nodes in τ_e and τ_r. (As explained above, a node u in τ_e acts as a proxy for a set of vertices in a particular callsite and $\eta(u)$ identifies this set.) Given η, define $\hat{\eta}$ as $\lambda X. \bigcup_{u \in X} \eta(u)$. We then define $\tau_e \langle\!\langle \tau_r, \eta \rangle\!\rangle$ to be $(\mathsf{EV}', \mathsf{EE}', \pi', \mathsf{IV}', \mathsf{IE}', \sigma')$ where

$$\begin{aligned}
\mathsf{V}' &= (\mathsf{IV}_r \cup \mathsf{EV}_r) \cup \hat{\eta}(\mathsf{IV}_e \cup \mathsf{EV}_e)\\
\mathsf{IV}' &= \mathsf{V}' \cap (\mathsf{IV}_r \cup \mathsf{IV}_e)\\
\mathsf{EV}' &= \mathsf{V}' \cap (\mathsf{EV}_r \cup \mathsf{EV}_e)\\
\mathsf{IE}' &= \mathsf{IE}_r \cup \{\langle v_1, f, v_2\rangle \mid \langle u, f, v\rangle \in \mathsf{IE}_e, v_1 \in \eta(u), v_2 \in \eta(v)\}\\
\mathsf{EE}' &= \mathsf{EE}_r \cup \{\langle u', f, v\rangle \mid \langle u, f, v\rangle \in \mathsf{EE}_e, u' \in \eta(u), \mathit{escapes}(u')\}\\
\pi' &= \pi_r\\
\sigma' &= \lambda x.\ x \in \mathit{Globals} \rightarrow \hat{\eta}(\sigma_e(x)) \mid x \in \mathit{Locals} \cup \mathit{Params} \rightarrow \sigma_r(x)\\
\mathit{escapes}(v) &\equiv \exists u \in \mathit{range}(\pi').v \text{ is reachable from } u \text{ via } \mathsf{IE}' \cup \mathsf{EE}' \text{ edges}
\end{aligned}$$

The predicate "escapes(u')" used in the above definition is recursively dependent on the graph τ' being constructed: it checks if u' is reachable from any of the parameter nodes in the graph being constructed. Thus, this leads to an iterative process for adding edges to the graph being constructed, as more escaping nodes are identified.

We now show how the node mapping function η is determined, given the transformers τ_e and τ_r. The function η is defined to be the least fixed point of the set of following constraints over the variable μ. (Here, μ_1 is said to be less than μ_2 iff $\mu_1(u) \subseteq \mu_2(u)$ for all u.) Let a_i denote the actual argument corresponding to the formal argument $\mathit{Param}(i)$.

$$x \in \mathsf{IV}_e \Rightarrow x \in \mu(x) \tag{13}$$

$$x \in \pi_e(\mathit{Param}(i)) \Rightarrow \sigma_r(a_i) \subseteq \mu(x) \tag{14}$$

$$x \in \pi_e(v) \wedge v \in \mathit{Globals} \Rightarrow \sigma_r(v) \subseteq \mu(x) \tag{15}$$

$$\langle u, f, v\rangle \in \mathsf{EE}_e, u' \in \mu(u), \langle u', f, v'\rangle \in \mathsf{IE}_r \Rightarrow v' \in \mu(v) \tag{16}$$

$$\langle u, f, v\rangle \in \mathsf{EE}_e, \mu(u) \cap \mu(u') \neq \emptyset, \langle u', f, v'\rangle \in \mathsf{IE}_e \Rightarrow \mu(v') \subseteq \mu(v) \tag{17}$$

$$\langle u, f, v\rangle \in \mathsf{EE}_e, \mu(u) \cap \mathit{Escaping}(\tau_e\langle\!\langle \tau_r, \mu\rangle\!\rangle) \neq \emptyset \Rightarrow v \in \mu(v) \tag{18}$$

In WSR analysis, rule (17) has one more pre-condition, namely ($u \neq u' \vee u \in \mathsf{EV}_e$). This extra condition may result in a more precise node mapping function but requires a similar change to the definition of the concretization function γ_T.

Abstract Fixed Point Computation. The collection of equations 9-12 can be viewed as a single equation $\vartheta = F^\sharp(\vartheta)$, where $F^\sharp$ is a function from $VE \mapsto \mathcal{F}_a$ to itself. Let $\bot$ denote $\lambda x.(\{\}, \{\}, \lambda v.\{\}, \{\}, \{\}, \lambda v.\{\})$. The analysis iteratively computes the sequence of values ${F^\sharp}^i(\bot)$ and terminates when ${F^\sharp}^i(\bot) = {F^\sharp}^{i+1}(\bot$). We define $[\![P]\!]_a$ (the summary for a procedure P) to be the value of $\varphi_{exit(P)}$ in the final solution.

Correctness and Termination. With this formulation, correctness and termination of the analysis follow in the standard way. Correctness follows by establishing that $F^\sharp$ is a sound approximation of $F^\natural$, which follows from the following lemma that the corresponding components of $F^\sharp$ are sound approximations of the corresponding components of $F^\natural$. As usual, we say that a concrete value $f \in \mathcal{F}_c$ is *correctly represented* by an abstract value $\tau \in \mathcal{F}_a$, denoted $f \sim \tau$, iff $f \sqsubseteq_c \gamma_T(\tau)$.

Lemma 2. *(a)* $\lambda g.\{g\} \sim \mathsf{ID}$
(b) For every primitive statement S (other than a procedure call), $[\![S]\!]_a$ is a sound approximation of $[\![S]\!]_c$: if $f \sim \tau$, then $f \circ [\![S]\!]_c \sim [\![S]\!]_a(\tau)$.
(c) $\sqcup_{co}$ is a sound approximation of $\sqcup_c$: if $f_1 \sim \tau_1$ and $f_2 \sim \tau_2$, then $(f_1 \sqcup_c f_2) \sim (\tau_1 \sqcup_{co} \tau_2)$.
(d) if $f_1 \sim \tau_1$ and $f_2 \sim \tau_2$, then $f_2 \circ CallReturn_S(f_1) \sim \tau_1\langle\!\langle\tau_2\rangle\!\rangle_a^S$.

Lemma 2 implies the following soundness theorem in the standard way (e.g., see Proposition 4.3 of [5]).

Theorem 1. *The computed procedure summaries are correct. (For every procedure P, $[\![P]\!]_c \sim [\![P]\!]_a$.)*

Termination follows by establishing that $F^\sharp$ is monotonic with respect to $\sqsubseteq^*_{co}$, since $\mathcal{F}_a$ has only finite height $\sqsubseteq_{co}$-chains. Proofs of all results appear in [11].

4 Optimizations

We have implemented the WSR analysis for .NET binaries. More details about the implementation and how we deal with language features absent in the core language used in our formalization appear in [11]. In this section we describe three optimizations for the analysis that were motivated by our implementation experience. We do not describe optimizations already discussed by WSR in [19] and [17]. We present an empirical evaluation of the impact of these optimizations on the scalability and the precision of the purity analysis in the experimental evaluation section.

Optimization 1: Node Merging. Informally, we define node merging as an operation that replaces a set of nodes $\{n_1, n_2 \ldots n_m\}$ by a single node n_{rep} such that any predecessor or successor of the nodes $n_1, n_2, \ldots, n_m$ becomes, respectively, a predecessor or successor of n_{rep}. While merging nodes seems like a natural heuristic for improving efficiency, it does introduce some subtle issues and challenges. The intuition for merging nodes arises from their use in the context of heap analyses where graphs represent sets of concrete states. However, in our context, graphs represent state transformers. We now present some results that help establish the correctness of this optimization.

We now extend the notion of graph embedding to transformer graphs. Given $\tau_1 = (\mathsf{EV}_1, \mathsf{EE}_1, \pi_1, \mathsf{IV}_1, \mathsf{IE}_1, \sigma_1)$ and $\tau_2 = (\mathsf{EV}_2, \mathsf{EE}_2, \pi_2, \mathsf{IV}_2, \mathsf{IE}_2, \sigma_2)$, we say that $\tau_1 \preceq \tau_2$ iff there exists a function $h : (\mathsf{IV}_1 \cup \mathsf{EV}_1) \mapsto (\mathsf{IV}_2 \cup \mathsf{EV}_2)$ such that: for every internal (respectively, external) node x in τ_1 , $h(x)$ is an internal (respectively, external) node; for every internal (respectively, external) edge $\langle x, f, y\rangle$ in τ_1,

$\langle h(x), f, h(y)\rangle$ is an internal (respectively, external) edge in τ_2, for every variable $\mathtt{x}$, $\hat{h}(\sigma_1(\mathtt{x})) \subseteq \sigma_2(\mathtt{x})$ and $\hat{h}(\pi_1(\mathtt{x})) \subseteq \pi_2(\mathtt{x})$ where $\hat{h}(Z) = \{h(u) \mid u \in Z\}$.

Node merging produces an embedding. Assume that we are given an equivalence relation $\simeq$ on the nodes of a transformer graph τ (such that no internal nodes are equivalent to external nodes). We define the transformer graph $\tau/\simeq$ to be the transformer graph obtained by replacing every node u by a unique representative of its $\simeq$-equivalence class in every component of τ.

Lemma 3. *(a) $\preceq$ is a pre-order. (b) γ_T is monotonic with respect to $\preceq$: i.e., $\forall \tau_a, \tau_b \in \mathcal{F}_a . \tau_a \preceq \tau_b \Rightarrow \gamma_T(\tau_a) \sqsubseteq_c \gamma_T(\tau_b)$. (c) $\tau \preceq (\tau/\simeq)$.*

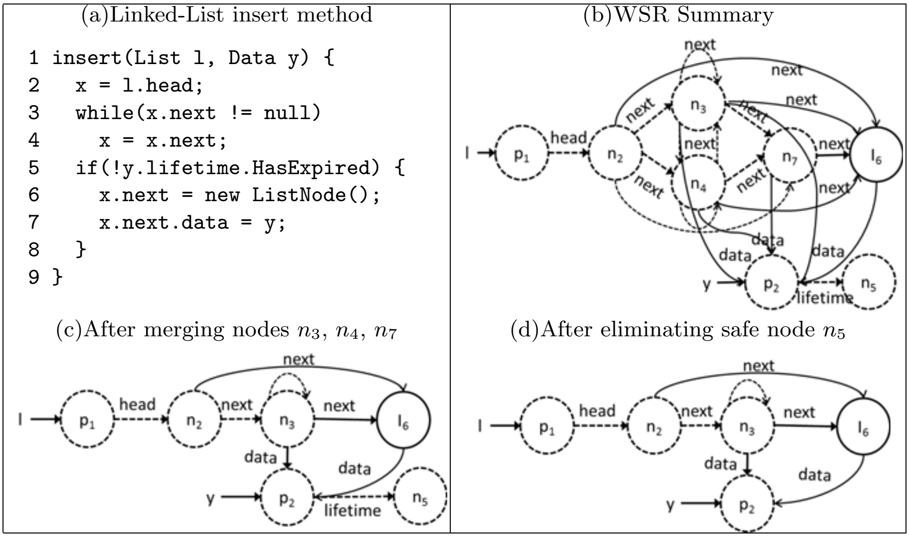
Assume that we wish to replace a transformer graph τ by a graph $\tau/\simeq$ at some point during the analysis (perhaps by incorporating this into one of the abstract operations). Our earlier correctness argument still remains valid (since if $f \sim \tau_1 \preceq \tau_2$, then $f \sim \tau_2$).

However, this optimization impacts the termination argument because we do not have $\tau \sqsubseteq_{co} (\tau/\simeq)$. Indeed, our initial implementation of the optimization did not terminate for one program because the computation ended up with a cycle of equivalent, but different, transformers (in the sense of having the same concretization). Refining the implementation to ensure that once two nodes are chosen to be merged together, they are always merged together in all subsequent steps, guarantees termination. Technically, we enhance the domain to include an equivalence relation on nodes (representing the nodes currently merged together) and update the transformers accordingly. A suitably modified ordering relation ensures termination. Details are omitted due to space constraints, but this illustrated to us the value of the abstract interpretation formalism (see [11] for more details).

The main advantage of the node merging optimization is that it reduces the size of the transformer graph while every other transfer function increases the size of the transformer graphs. However, when used injudiciously, node merging can result in loss of precision. In our implementation we use a couple of heuristics to identify the set of nodes to be merged.

Given $\tau \in \mathcal{F}_a$ and $v_1, v_2 \in \mathsf{V}(\tau)$, we merge v_1, v_2 iff one of the two conditions hold (a) $v_1, v_2 \in \mathsf{EV}(\tau)$ and $\exists u \in \mathsf{V}(\tau)$ s.t. $\langle u, f, v_1\rangle \in \mathsf{EE}(\tau)$ and $\langle u, f, v_2\rangle \in \mathsf{EE}(\tau)$ for some field f or (b) $v_1, v_2 \in \mathsf{IV}(\tau)$ and $\exists u \in \mathsf{V}(\tau)$ s.t. $\langle u, f, v_1\rangle \in \mathsf{IE}(\tau)$ and $\langle u, f, v_2\rangle \in \mathsf{IE}(\tau)$ for some field f.

In the WSR analysis, an external edge $\langle u, f, v\rangle$ on an escaping node u is often used to identify objects that $u.f$ may point-to in the state before the call to the method (i.e, pre-state). However, having two external edges with the same source and same field serves no additional purpose. Our first heuristic eliminates such duplicate external edges, which may be produced, e.g., by multiple reads "`x.f`", where `x` is a formal parameter, of the same field of a pre-state object inside a method or its transitive callees. Our second heuristic addresses a similar problem that might arise due to multiple writes to the same field of an internal object inside a method or its transitive callees. Although, theoretically, the above two heuristics can result in loss of precision, it was not the case on most of the



(a)Linked-List insert method

```
1 insert(List l, Data y) {
2   x = l.head;
3   while(x.next != null)
4     x = x.next;
5   if(!y.lifetime.HasExpired) {
6     x.next = new ListNode();
7     x.next.data = y;
8   }
9 }
```

(b)WSR Summary

(c)After merging nodes n_3, n_4, n_7

(d)After eliminating safe node n_5

Fig. 4. Illustrative example for the optimizations

programs on which we ran our analysis (see experimental results section). We apply this node-merging optimization only at procedure exit (to the summary graph produced for the procedure).

Figure 4 shows an illustration of this optimization. Figure 4(a) shows a simple procedure that appends an element to a linked list. Figure 4(b) shows the WSR summary graph that would result by the straight forward application of the transfer functions presented in the paper. Figure 4(c) shows the impact of applying the node-merging optimization on the WSR summary shown in Figure 4(b). In the WSR summary, it can be seen that the external node n_2 has three outgoing external edges on the field *next* that end at nodes n_3, n_4 and n_7. This is due to the reads of the field *next* in the line numbers 3, 4 and 7. As shown in Figure 4(b) the blow-up due to these redundant edges is substantial (even in this small example). Figure 4(c) shows the transformer graph that results after merging the nodes n_3, n_4 and n_7 that are identified as equivalent by our heuristics. Let the transformer graphs shown in Figure 4(b) and Figure 4(c) be τ_a and τ_b respectively. It can be verified that $\gamma(\tau_a) = \gamma(\tau_b)$.

Optimization 2: Summary Merging. Though the analysis described earlier does not consider virtual method calls, our implementation does handle them (explained in [11]). Briefly, a virtual method call is modelled as a conditional call to one of the various possible implementation methods. Let the transformer graph before and after the virtual method call statement be τ_{in} and τ_{out} respectively. Let the summaries of the possible targets of the call be $\tau_1, \tau_2, \ldots \tau_n$. In the unoptimized approach, $\tau_{out} = \tau_1\langle\langle\tau_{in}\rangle\rangle \sqcup_{co} \ldots \sqcup_{co} \tau_n\langle\langle\tau_{in}\rangle\rangle$. This optimization constructs a single summary that over-approximates all the callee summaries, as $\tau_{merge} = \tau_1 \sqcup_{co} \ldots \sqcup_{co} \tau_n$ and computes τ_{out} as $\tau_{merge}\langle\langle\tau_{in}\rangle\rangle$. Since each

$\tau_i \preceq \tau_{merge}$ (in fact, $\tau_i \sqsubseteq_{co} \tau_{merge}$), τ_{merge} is a safe over-approximation of the summaries of all callees. Once the graph τ_{merge} is constructed it is cached and reused when the virtual method call instruction is re-encountered during the fix-point computation (provided the targets of the virtual method call do not change across iterations and their summaries do not change). We further apply node merging to τ_{merge} to obtain τ_{mo} which is used instead of τ_{merge}.

Optimization 3: Safe Node Elimination. This optimization identifies certain external nodes that can be discarded from a method's summary without affecting correctness. As motivation, consider a method *Set::Contains*. This method does not mutate the caller's state, but its summary includes several external nodes that capture the "reads" of the method. These extraneous nodes make subsequent operations more expensive. Let m be a method with a summary τ. An external vertex ev is safe in τ iff it satisfies the following conditions for every vertex v transitively reachable from ev: (a) v is not modified by the procedure, and (b) No internal edge in τ ends at v and there exists no variable t such that $v \in \sigma(t)$. (We track modifications of nodes with an extra boolean attached to nodes.) Let $removeSafeNodes(\tau)$ denote transformer obtained by deleting all safe nodes in τ. We can show that $\gamma_T(removeSafeNodes(\tau)) = \gamma_T(\tau)$. Like node merging we perform this optimization only at method exits. Figure 4(d) shows the transformer graph that would result after eliminating safe nodes from the transformer graph shown in Figure 4(c).

5 Empirical Evaluation

We implemented the purity analysis along with the optimizations using *Phoenix* analysis framework for .NET binaries [12]. In our implementation, summary computation is performed using an intra-procedural *flow-insensitive* analysis using the transfer functions described in Figure 3. We chose a flow-insensitive analysis due to the prohibitively large memory requirements of a flow-sensitive analysis when run on large libraries. We believe that the optimizations that we propose will have a bigger impact on the scalability of a flow-sensitive analysis.

Fig. 5 shows the benchmarks used in our evaluation. All benchmarks (except *mscorlib.dll* and *System.dll*) are open source C# libraries[4]. We carried out our experiments on a 2.83 GHz, 4 core, 64 bit Intel Xeon CPU running Windows Server 2008 with 16GB RAM.

We ran our implementation on all benchmarks in six different configurations (except *QuickGraph* which was run on three configurations only) to evaluate our optimizations: (a) base WSR analysis without any optimizations (*base*) (b) base analysis with summary merging (*base+sm*) (c) base analysis with node merging (*base+nm*) (d) base analysis with summary and node merging (*base+nsm*) (e) base analysis with safe node elimination (*base+sf*) (f) base analysis with all optimizations (*base+all*). We impose a time limit of 3 hours for the analysis of each program (except *QuickGraph* where we used a time limit of 8 hours).

Benchmark	**LOC**	**Description**
DocX (*dx*)	10K	library for manipulating Word 2007 files
Facebook APIs (*fb*)	21K	library for integrating with Facebook.
Dynamic data display (*ddd*)	25K	real-time data visualization tool
SharpMap (*sm*)	26K	Geospatial application framework
Quickgraph (*qg*)	34K	Graph Data structures and Algorithms
PDfsharp (*pdf*)	96K	library for processing PDF documents
DotSpatial (*ds*)	220K	libraries for manipulating Geospatial data
mscorlib (*ms*)	Unknown	Core C# library
System (*sys*)	Unknown	Core C# library

Fig. 5. benchmark programs

Benchmarks	*dx*	*fb*	*ddd*	*pdf*	*sm*	*ds*	*ms*	*sys*	*qg*
# of methods	612	4112	2266	3883	1466	10810	2963	698	3380
Pure methods	340	1924	1370	1515	934	5699	1979	411	2152
time(s)									
base	21	52	4696	5088	∞	∞	108	17	∞
base+sf	19	46	3972	2914	∞	∞	56	16	–
base+sm	6	14	3244	4637	7009	∞	54	5	∞
base+nm	20	46	58	125	615	963	21	16	–
base+nsm	5	9	26	79	181	251	13	4	–
base+all	5	8	23	76	179	232	12	4	21718
memory(MB)									
base	313	478	1937	1502	∞	∞	608	387	∞
base+sf	313	460	1836	1136	∞	∞	545	390	–
base+sm	313	478	1937	1508	369	∞	589	390	∞
base+nm	296	460	427	535	356	568	515	387	–
base+nsm	296	461	411	569	369	568	514	390	–
base+all	296	446	410	550	356	568	497	390	703

Fig. 6. Results of analysing the benchmarks in six configurations

Fig. 6 shows the execution time and memory consumption of our implementation. Runs that exceed the time limit were terminated and their times are listed as ∞. The number of methods classified as pure were same for all configurations (that terminated) for all benchmarks.

The results show that for several benchmarks, node merging drastically reduces analysis time. The other optimizations also reduce the analysis time, though not as dramatically as node merging. Fig. 7 provides insights into the reasons for this improvement by illustrating the correlation between analysis time and number of duplicate edges in the summary. A point (x, y) in the graph indicates that y percentage of analysis time was spent on procedures whose summaries had, on average, at least x outgoing edges per vertex that are labelled by the same field. The benchmarks that benefited from the node merging optimization (viz. SharpMap, PDFSharp, Dynamic Data Display, DotSpatial) spend a

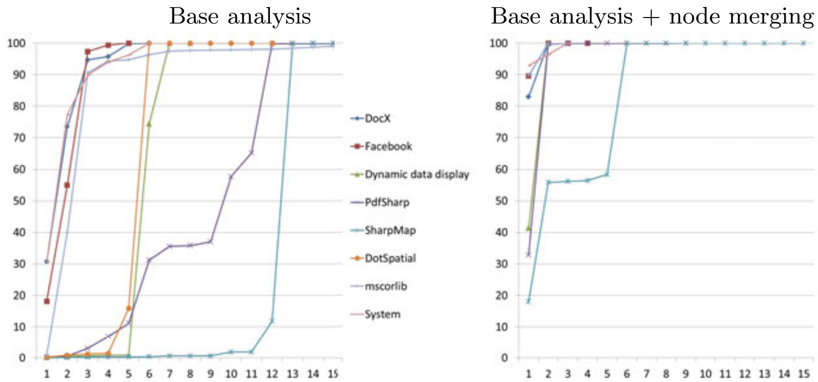


Fig. 7. Number duplicate edges in the summary graph Vs percentage time taken to compute the summary

large fraction of the analysis time (approx. 90% of the time) on summaries that have average number of duplicate edges per vertex above 4. The graph on the right hand side plots the same metrics when node merging is enabled. It can be seen that node merging is quite effective in reducing the duplicate edges and hence also reduces analysis time.

6 Related Work

Modular Pointer Analyses. The Whaley-Rinard analysis [19], which is the core of Salcianu-Rinard's purity analysis [17], is one of several modular pointer analyses that have been proposed, such as [2] and [3]. Modular pointer analyses offer the promise of scalability to large applications, but are quite complex to understand and implement. We believe that an abstract interpretation formulation of such modular analyses are valuable as they make them accessible to a larger audience and simplify reasoning about variations and modifications of the algorithm. We are not aware of any previous abstract interpretation formulation of a modular pointer analysis. Our formulation also connects the WSR approach to Sharir-Pnueli's functional approach to interprocedural analysis [18].

Compositional Shape Analyses. Calcagno *et al.* [1] and Gulavani *et al.* [7] present separation-logic based compositional approaches to shape analysis. They perform more precise analysis but compute Hoare triples, which correspond to conditional summaries: summaries which are valid only in states that satisfy the precondition of the Hoare triple. These summaries typically incorporate significant "non-aliasing" conditions in the precondition. Modular pointer analyses such as WSR have somewhat different goals. They are less precise, but more scalable and produce summaries that can be used in any input state.

Parametric Shape Analyses. TVLA [15] is a parametric abstract interpretation that has been used to formalize a number of heap and shape analyses. The WSR analysis and our formalization seem closely related to the relational approach to interprocedural shape analysis presented by Jeannet *et al.* [9]. The Jeannet *et al.*approach shows how the abstract shape graphs of TVLA can be used to represent abstract graph transformers (using a double vocabulary), which is used for modular interprocedural analysis. Rinetzky *et al.* [14] present a tabulation-based approach to interprocedural heap analysis of cutpoint-free programs (which imposes certain restrictions on aliasing). (While the WSR analysis computes a procedure summary that can be reused at any callsite, the tabulation approach may analyze a procedure multiple times, but reuses analysis results at different callsites if the "input heap" is the same.) However, there are interesting similarities and connections between the WSR approach and the Rinetzky *et al.* approach to merging "graphs" from the callee and the caller.

Modularity In Interprocedural Analysis. While the WSR analysis is modular in the absence of recursion, recursive procedures must be analyzed together. Our experience has shown that large strongly connected components of procedures in the call-graph can be a bottleneck in analyzing large libraries. An interesting direction for future work is to explore techniques that can be used to achieve modularity even in the presence of recursion, e.g., see [6].

References

1. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL, pp. 289–300 (2009)
2. Chatterjee, R., Ryder, B.G., Landi, W.A.: Relevant context inference. In: POPL, pp. 133–146 (1999)
3. Cheng, B.C., Hwu, W.M.W.: Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In: PLDI, pp. 57–69 (2000)
4. Codeplex (March 2011), http://www.codeplex.com
5. Cousot, P., Cousot, R.: Abstract interpretation frameworks. J. Log. Comput. 2(4), 511–547 (1992)
6. Cousot, P., Cousot, R.: Modular static program analysis. In: CC 2002. LNCS, vol. 2304, pp. 159–178. Springer, Heidelberg (2002)
7. Gulavani, B.S., Chakraborty, S., Ramalingam, G., Nori, A.V.: Bottom-up shape analysis. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 188–204. Springer, Heidelberg (2009)
8. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: SYNERGY: a new algorithm for property checking. In: Robshaw, M.J.B. (ed.) FSE 2006. LNCS, vol. 4047, pp. 117–127. Springer, Heidelberg (2006)
9. Jeannet, B., Loginov, A., Reps, T., Sagiv, M.: A relational approach to interprocedural shape analysis. ACM Trans. Program. Lang. Syst. 32, 5:1–5:52 (2010), http://doi.acm.org/10.1145/1667048.1667050
10. Knoop, J., Steffen, B.: The interprocedural coincidence theorem. In: Pfahler, P., Kastens, U. (eds.) CC 1992. LNCS, vol. 641, pp. 125–140. Springer, Heidelberg (1992)
11. Madhavan, R., Ramalingam, G., Vaswani, K.: Purity analysis: An abstract interpretation formulation. Tech. rep., Microsoft Research, India (forthcoming)
12. Phoenix (March 2011), https://connect.microsoft.com/Phoenix
13. Prabhu, P., Ramalingam, G., Vaswani, K.: Safe programmable speculative parallelism. In: PLDI, pp. 50–61 (2010)

14. Rinetzky, N., Sagiv, M., Yahav, E.: Interprocedural shape analysis for cutpoint-free programs. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 284–302. Springer, Heidelberg (2005)
15. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: POPL, pp. 105–118 (1999)
16. Salcianu, A.D.: Pointer Analysis and its Applications for Java Programs. Master's thesis, Massachusetts institute of technology (2001)
17. Salcianu, A.D., Rinard, M.C.: Purity and side effect analysis for java programs. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 199–215. Springer, Heidelberg (2005)
18. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Program Flow Analysis: Theory and Applications, pp. 189–234 (1981)
19. Whaley, J., Rinard, M.C.: Compositional pointer and escape analysis for java programs. In: OOPSLA, pp. 187–206 (1999)

The Complexity of Abduction for Separated Heap Abstractions

Nikos Gorogiannis, Max Kanovich, and Peter W. O'Hearn

Queen Mary University of London

Abstract. Abduction, the problem of discovering hypotheses that support a conclusion, has mainly been studied in the context of philosophical logic and Artificial Intelligence. Recently, it was used in a compositional program analysis based on separation logic that discovers (partial) pre/post specifications for un-annotated code which approximates memory requirements. Although promising practical results have been obtained, completeness issues and the computational hardness of the problem have not been studied. We consider a fragment of separation logic that is representative of applications in program analysis, and we study the complexity of searching for feasible solutions to abduction. We show that standard entailment is decidable in polynomial time, while abduction ranges from NP-complete to polynomial time for different sub-problems.

1 Introduction

Abductive inference is a mode of reasoning that concerns generation of new hypotheses [25]. Abduction has attracted attention in Artificial Intelligence (e.g., [24]), based on the idea that humans perform abduction when reasoning about the world, such as when a doctor looking at a collection of symptoms hypothesizes a cause which explains them.

Similarly, when a programmer tries to understand a piece of code, he or she makes hypotheses as well as deductions. If you look at the C code for traversing a cyclic linked list, you might hypothesize that an assertion describing a cyclic list should be part of the precondition, else one would obtain a memory error, and you might even discover this *from the code itself* rather than by communication from the program's designer. In separation logic, a specialized logic for computer memory, the abduction problem – given A and B, find X where the separating conjunction $A * X$ is consistent and $A * X$ entails B – takes on a spatial connotation where X describes "new" or "missing" memory, not available in the part of memory described by A. Recent work has used abductive inference for separation logic to construct an automatic program analysis which partly mimics, for a restricted collection of assertions describing memory-usage requirements of procedures, the combined abductive-deductive-inductive reasoning that programmers employ when approaching bare code [6]. Abduction is used in the generation of preconditions, after which forwards analysis can be used to obtain a postcondition and, hence, a true Hoare triple for the procedure, without consulting the procedure's calling context, resulting in a compositional analysis.

E. Yahav (Ed.): SAS 2011, LNCS 6887, pp. 25–42, 2011.

Compositionality – that the analysis result of a whole is computed from the analysis results of its parts – has well-known benefits in program analysis [9], including the ability to analyze incomplete programs (e.g., programs as they are being written) and increased potential to scale. Abductive inference has enabled a boost in the level of automation in shape analysis (e.g., [26,13]) – an expensive "deep memory" analysis which involves discovering data structures of unbounded depth in the heap. The Abductor academic prototype tool has been applied to several open-source projects in the hundreds of thousands of LOC [11], and Infer is an industrial tool which incorporates these and other ideas [5].

Other applications of abduction for separation logic include the analysis of concurrent programs [7], memory leaks [12], abduction for functional correctness rather than just memory safety [17], and discovering specifications of unknown procedures [22]. The latter is somewhat reminiscent of the (to our knowledge) first use of abduction in program analysis, a top-down method that infers constraints on literals in a logic program starting from a specification of a top-level program [16]. In contrast, Abductor works bottom-up, obtaining specs for procedures from specs of their callees (it would evidently be valuable to mix the two approaches). A seemingly unrelated use of abduction in program analysis is in under-approximation of logical operators such as conjunction and disjunction in abstract domains with quantification [18].

While the potential applications are perhaps encouraging, the abduction problem for separated heap abstractions has not been investigated thoroughly from a theoretical point of view. The proof procedures used are pragmatically motivated and sound but demonstrably incomplete, and questions concerning complexity or the existence of complete procedures have not been addressed. Our purpose in this paper is to consider complexity questions (taking completeness as a requirement) for the abduction problem for a fragment of logic representative of that used in program analysis.

In the context of classical logic, abduction has been studied extensively and there are several results about its algorithmic properties when using, for example, different fragments of propositional logic as the base language [14,10]. However, the results do not carry over to our problem because the special abstract domains used in shape analyzers are different in flavour from propositional logic. For example, the use of variables and equalities and disequalities in separated heap abstractions raise particular problems. Furthermore, understanding the interaction between heap-reachability and separation is subtle but essential.

The contents of the paper are as follows. In Section 2 we define the restricted separation logic formulae we use, called 'symbolic heaps' [2,13], which include a basic 'points-to' predicate, and an inductive predicate for describing linked-list segments. The separated abduction problem is defined in Section 3 along with a 'relaxed' version of the problem, often used in program analysis. Section 4 shows that entailment is in PTIME and contains an interpolation-like result which bounds the sizes of solutions that must be considered in abduction. Section 5 establishes that when lists are present both the general and relaxed problems are NP-complete. Section 6 shows that the abduction problem is NP-complete when the formulae have only points-to predicates, and that the 'relaxed' version of the problem can be solved in polynomial time.

2 Preliminaries

This section records background material on separated heap abstractions [2,13].

2.1 Syntax of Separated Heap Abstractions

Let $\mathtt{Var}$ be a countable set of variables. The set of terms is simply $\mathsf{Terms} = \mathtt{Var} \cup \{\mathtt{nil}\}$ where $\mathtt{nil} \notin \mathtt{Var}$. Spatial predicates P, pure formulae Π and spatial formulae Σ are defined as follows, where x, y are terms.

$$\begin{aligned} P(x,y) &::= x \mapsto y \mid \mathtt{ls}(x,y) \\ \Pi &::= \Pi \wedge \Pi \mid x = y \mid x \neq y \\ \Sigma &::= \Sigma * \Sigma \mid P(x,y) \mid \mathtt{emp} \mid \mathtt{true} \end{aligned}$$

A formula in one of the forms: $\Pi \wedge \Sigma$, Π, or Σ is called a *symbolic heap*. We employ $\equiv$ to denote syntactic equality of two expressions modulo commutativity of $\wedge$, $*$, and symmetry of $=$ and $\neq$. We will say that the term x is an *L-value* in a formula A if there is a term y such that the spatial predicate $P(x, y)$ is in A.

The separating conjunction of two symbolic heaps is defined as follows.

$$(\Pi_A \wedge \Sigma_A) * (\Pi_B \wedge \Sigma_B) = (\Pi_A \wedge \Pi_B) \wedge (\Sigma_A * \Sigma_B)$$

This definition is, in fact, an equivalence in general separation logic.

The formula $\mathtt{ls}(y, z)$ expresses that the heap consists of a non-empty acyclic path from y to z, and that z is not an allocated cell in the heap. The formula $x \mapsto y$ describes a singleton heap in which x is allocated and has contents y. Cycles can be expressed with compound formulae. For instance, $\mathtt{ls}(y, x) * \mathtt{ls}(x, y)$ describes two acyclic non-empty linked lists which together form a cycle: this is the kind of structure sometimes used in cyclic buffer programs.

As always in program analysis, what *cannot* be said is important for allowing efficient algorithms for consistency and entailment checking. General separation logic, which allows $*$ and its adjoint $-\!\!*$ to be combined with all boolean connectives, is undecidable even at the propositional level [4]. In the fragment here we cannot express, e.g., that there are not two separate lists in the heap, or that the data elements held in a list are sorted. We can describe memory safety properties of linked-list programs (that data structures are well formed) but not functional correctness (e.g., that a list insertion procedure actually inserts). As we will see, we obtain a polynomial-time algorithm for entailment (hence, consistency). In more expressive decidable heap logics (e.g., [23,27,21,3]) these problems can range from PSPACE to even non-elementary complexity.

2.2 Semantics

The logic is based on a model of *heap partitioning*.

Definition 2.1. *Stack-and-heap models* are defined as pairs (s, h), where s (the *stack*) is a mapping from variables $\mathtt{Var}$ to values $\mathtt{Val}$, and h (the *heap*) is a finite partial function from an infinite L to RV (L-values to R-values in Strachey's terminology). Here we take RV to be $L \cup \{\mathtt{nil}\}$ where $\mathtt{nil} \notin L$. The composition $h_1 \circ h_2$ is the union of h_1 and h_2 if $\mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) = \emptyset$, else undefined.

The value $\mathtt{nil}$ plays the role of a special never-to-be allocated pointer which is useful, e.g., for terminating linked lists. The heaps here have at most one successor for any node, reflecting our focus on linked lists rather than trees or graphs in the symbolic heaps that we study. (Eventually, we would like to consider general inductive definitions; they are easy to consider semantically, but extremely challenging, and beyond our scope, for decidability of entailment.)

We will use stacks as functions and write $s(x)$ for the value of a variable x. We extend this notation to include $\mathtt{nil}$: $s(\mathtt{nil}) = \mathtt{nil}$. Whenever $s(x) = a$ and $s(y) = b$, the *spatial* predicate $x \mapsto y$ is true in the one-cell heap of the form $a \rightarrow \boxed{\text{b}}$, or $\overset{a}{\bullet} \longrightarrow \overset{b}{\bullet}$, which depicts that the 'location' a contains the value b.

The formal definition of the semantics is as follows.

Definition 2.2. Given any (s, h) and formula A, we define the forcing relation $(s, h) \vDash A$ by induction on A (see [2]):

$(s, h) \vDash \mathtt{emp}$ iff $h = []$ is the empty heap
$(s, h) \vDash A * B$ iff $\exists h_1, h_2.\ h = h_1 \circ h_2$ and $(s, h_1) \vDash A$ and $(s, h_2) \vDash B$,
$(s, h) \vDash A \wedge B$ iff $(s, h) \vDash A$ and $(s, h) \vDash B$,
$(s, h) \vDash \mathtt{true}$ always,
$(s, h) \vDash (x = y)$ iff $s(x) = s(y)$,
$(s, h) \vDash (x \neq y)$ iff $s(x) \neq s(y)$,
$(s, h) \vDash x \mapsto y$ iff $\mathrm{dom}(h) = \{s(x)\}$ and $h(s(x)) = s(y)$,
$(s, h) \vDash \mathtt{ls}(x, y)$ iff for some $n \geq 1$, $(s, h) \vDash \mathtt{ls}^{(n)}(x, y)$,
$(s, h) \vDash \mathtt{ls}^{(n)}(x, y)$ iff $|\,\mathrm{dom}(h)| = n$, and there is a chain $a_0, \ldots, a_n$, *with no repetitions*, such that $h(a_0) = a_1, \ldots, h(a_{n-1}) = a_n$, where $a_0 = s(x)$, $a_n = s(y)$ and $a_n \neq a_0$ (notice that $s(y) \notin \mathrm{dom}(h)$).

Remark 2.3. A *precise* formula [8] cuts out a unique piece of heap, making the non-deterministic $\exists$-selection in the semantics of $*$ become deterministic. For instance, if $(s, h) \vDash \mathtt{ls}(x, y) * B$, then h is *uniquely* split into h_1, h_2 so that $(s, h_1) \vDash \mathtt{ls}(x, y)$ and $(s, h_2) \vDash B$, where h_1 is defined as an acyclic path from $s(x)$ to $s(y)$. In this fragment, any formula not containing $\mathtt{true}$ is precise.

As usual, we say that a formula A is *consistent* if $(s, h) \vDash A$ for some (s, h). A sequent $A \vDash B$ is called *valid* if for any model (s, h) such that $(s, h) \vDash A$ we have $(s, h) \vDash B$. Finally, we call a formula *explicit* if it syntactically contains all equalities and disequalities it entails.
We shall use the inference rules below [2].

$$A \vDash B \implies A * C \vDash B * C \qquad (*\text{-Intr})$$

$$x = y \wedge A \vDash B \iff A[x/y] \vDash B[x/y] \qquad (\text{Subst})$$

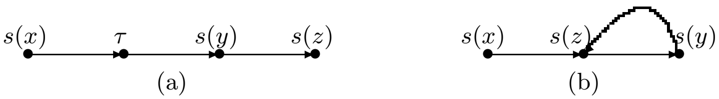


Fig. 1. (a) The fully acyclic heap. (b) The scorpion-like heap which destroys the validity of $(z \neq x) \wedge (z \neq y) \wedge \mathtt{ls}(x,y) * y \mapsto z \vDash \mathtt{ls}(x,z)$.

Rule ∗-Intr expresses the monotonicity of ∗ w.r.t ⊨. Rule Subst is a standard substitution principle.

Sample True and False Entailments

$x \mapsto x' * y \mapsto y' \vDash x \neq y$	$x \mapsto x' * \mathtt{ls}(x',y) \nvDash \mathtt{ls}(x,y)$
$x \mapsto x' * x \mapsto y' \vDash x \neq x$	$x \mapsto x' * \mathtt{ls}(x',y) * y \mapsto z \vDash \mathtt{ls}(x,y) * y \mapsto z$
$\mathtt{ls}(x,x') * \mathtt{ls}(y,y') \vDash x \neq y$	

The three examples on the left illustrate the anti-aliasing or separating properties of ∗, where the two on the right illustrate issues to be taken into account in rules for appending onto the end of list segments (a point which is essential in completeness considerations [2]). Appending a cell to the head of a list is always valid, as long as we make sure the end-points are distinct:

$$(z \neq x) \wedge x \mapsto y * \mathtt{ls}(y,z) \vDash \mathtt{ls}(x,z),$$

whereas appending a cell to the tail of a list generally is not valid. Although $s(z) \neq s(x)$ and $s(z) \neq s(y)$ in Fig 1(b), the dangling z stings an intermediate point τ in $\mathtt{ls}(x,y)$, so that

$$(z \neq x) \wedge (z \neq y) \wedge \mathtt{ls}(x,y) * y \mapsto z \nvDash \mathtt{ls}(x,z).$$

To provide validity, we have to 'freeze' the dangling z, for instance, with $\mathtt{ls}(z,v)$:

$$\mathtt{ls}(x,y) * \mathtt{ls}(y,z) * \mathtt{ls}(z,v) \vDash \mathtt{ls}(x,z) * \mathtt{ls}(z,v)$$

3 Separated Abduction Problems

Before giving the main definitions, it will be helpful to provide some context by a brief discussion of how abduction can be used in program analysis.

We consider a situation where each program operation has preconditions that must be met for the operation to succeed. A pointer dereferencing statement $\mathtt{x} \rightarrow \mathtt{next} = \mathtt{y}$ might have the assertion $x \mapsto x'$ as a precondition. A larger procedure might have preconditions tabulated as part of a 'procedure summary'. Abduction is used during a forwards-running analysis to find out what is missing from the current abstract state at a program point, compared to what is required by the operation, and this abduced information is used to build up an overall precondition for a body of code.

For example, suppose that the assertion

$\mathtt{ls}(x,\mathtt{nil}) * \mathtt{ls}(y,\mathtt{nil})$ is the precondition for a procedure (it might be a procedure to merge lists) and assume that we have the assertion $x \mapsto \mathtt{nil}$ at the call site of the procedure. Then solving

$$x \mapsto \mathtt{nil} * Y? \models \mathtt{ls}(x, \mathtt{nil}) * \mathtt{ls}(y, \mathtt{nil})$$

would tell us information we could add to the current state, in order to abstractly execute the procedure. An evident answer is $Y = \mathtt{ls}(y, \mathtt{nil})$ in this case.

Sometimes, there is additional material in the current state, not needed by the procedure; this unused portion of state is called the frame ([20], after the frame problem from Artificial Intelligence). Suppose $x \mapsto \mathtt{nil} * z \mapsto w * w \mapsto z$ is the current state and $\mathtt{ls}(x, \mathtt{nil}) * \mathtt{ls}(y, \mathtt{nil})$ is again the procedure precondition, then $z \mapsto w * w \mapsto z$ is the leftover part. In order to cater for leftovers, abduction is performed with `true` as a $*$-conjunct on the right. To see why, consider that

$$x \mapsto \mathtt{nil} * z \mapsto w * w \mapsto z * Y? \models \mathtt{ls}(x, \mathtt{nil}) * \mathtt{ls}(y, \mathtt{nil}).$$

has no consistent solution, since the cycle between z and w cannot form a part of a list from y to `nil`. However, $Y = \mathtt{ls}(y, \mathtt{nil})$ is indeed a solution for

$$x \mapsto \mathtt{nil} * z \mapsto w * w \mapsto z * Y? \models \mathtt{ls}(x, \mathtt{nil}) * \mathtt{ls}(y, \mathtt{nil}) * \mathtt{true}.$$

In the approach of [6] a separate mechanism, frame inference, is used after abduction, to percolate the frame from the precondition to the postcondition of an operation. We will refer to this special case of the abduction problem, with `true` on the right, as the *relaxed* abduction problem.

With this as background, we now give the definition of the main problems studied in the paper.

Definition 3.1. **(1)** We say that X is a solution to the abduction problem $A * X? \models B$ if $A * X$ is consistent and the sequent $A * X \models B$ is valid.

(2) We use $\mathrm{SAP}(\mapsto)$ to refer to the abduction problem restricted to formulae that have no `ls` predicate, and $\mathrm{SAP}(\mapsto, \mathtt{ls})$ for the general problem. The inputs are two symbolic heaps A and B. The output is 'yes' if there is a solution to the problem $A * X? \models B$, 'no' otherwise. $\mathrm{rSAP}(\mapsto)$ and $\mathrm{rSAP}(\mapsto, \mathtt{ls})$ refer to the specializations of these problems when B is required to include $*\mathtt{true}$.

We have formulated SAP as a decision problem, a yes/no problem. For applications to program analysis the analogous problem is a search problem: find a particular solution to the abduction problem $A * X? \models B$, or say that no solution exists. Clearly, the decision problem provides a lower bound for the search problem and as such, when considering NP-hardness we will focus on the decision problem. When considering upper bounds, we give algorithms for the search problem, and show membership of the decision problem in the appropriate class.

In general when defining abduction problems, the solutions are restricted to 'abducible' facts, which in classical logic are often conjunctions of literals. Here, the symbolic heaps are already in restricted form, which give us a separation logic analogue of the notion of abducible: $*$-conjunctions of points-to and list segment predicates, and $\wedge$-conjunctions of equalities and disequalities. Also, when studying abduction, one often makes a requirement that solutions be minimal in some sense. At least two criteria have been offered in the literature for minimality of SAP [6,22]. We do not study minimality here. Our principal results on NP-hardness apply as well to algorithms searching for minimal solutions as lower bounds, and we believe that our 'easiness' results concerning cases when polytime is achievable could carry over to minimality questions. We have concentrated on the more basic case of consistent solutions here, leaving minimality for the future.

4 Membership in NP

The main result of this section is the following, covering both SAP($\mapsto$, $\mathtt{ls}$) and rSAP($\mapsto$, $\mathtt{ls}$).

Theorem 4.1 (**NP upper bound**). *There is an algorithm, running in nondeterministic polynomial time, such that for any formulae A and B, it outputs a solution X to the abduction problem $A * X? \models B$, or says that no solution exists.*

Proof. Given A and B, let $\mathcal{Z} = \{z_1, z_2, .., z_n\}$ be the set of all variables and constants occurring in A and B. We define a set of *candidates* X in the following way. The spatial part Σ_X of X is defined as

$$x_1 \mapsto z_{i_1} \;*\; x_2 \mapsto z_{i_2} * \cdots * \; x_m \mapsto z_{i_m}$$

where *distinct* x_1, x_2,.., x_m are taken from $\mathcal{Z}$ and $\{z_{i_1}, z_{i_2}, .., z_{i_m}\} \subseteq \mathcal{Z}$ (for the sake of consistency we take x_1, x_2,.., x_m that are not L-values in A).

The pure part Π_X of X is defined as an $\wedge$-conjunction of formula of the form $(z_i = z_j)^{\varepsilon_{ij}}$, where $(z_i = z_j)^1$ stands for $(z_i = z_j)$, and $(z_i = z_j)^0$ stands for $(z_i \neq z_j)$.

The size of any *candidate* X is $\mathcal{O}(n^2)$ is quadratic in the size of A and B. Since consistency and entailment are in PTIME (Theorem 4.3 below), each *candidate* X can be checked in polynomial time as to whether it is a solution or not.

The Interpolation Theorem (Theorem 4.4 below) guarantees the completeness of our procedure. □

The gist of this argument is that we can check a candidate solution in polynomial time, and only polynomial-sized solutions need be considered (by Interpolation). The entailment procedure we use to check solutions relies essentially on our use of *necessarily non-empty* list segments (as in [13]). For formulae with *possibly-empty* list segments, which are sometimes used (e.g., [19]), we do not know if entailment can be decided in polynomial time; indeed, this has been an open question since symbolic heaps were introduced [2].

However, we can still find an NP upper bound for abduction, even if we consider possibly empty list segment predicates. The key idea is to consider *saturated solutions* only, i.e., solutions which, for any two variables contain either an equality or disequality between them. For candidate saturated solutions there is no empty/non-empty ambiguity, and we can fall back on the polytime entailment procedure below. Furthermore, a saturation can be guessed by an NP algorithm.

This remark is fleshed out in the following theorem.

Theorem 4.2 (**NP upper bound**).. *There is an algorithm, running in nondeterministic polynomial time, such that for any formulae A and B in the language extended with possibly-empty list segments, it outputs a particular solution X to the abduction problem $A * X? \models B$, or says that no solution exists.*

We now turn to the two results used in the proof of the above theorems.

Theorem 4.3 (Entailment/Consistency). *There is a sound and complete algorithm that decides $A \models B$ in polynomial time. As a consequence, consistency of symbolic heaps can also be decided in polynomial time.*

Proof Sketch. The main idea behind the algorithm is to turn the sequent $A \models B$ into an equi-valid sequent $A' \models B$ where all `ls` predicates in A have been converted to $\mapsto$ predicates in A'. A sequent of this form can be decided using "subtraction" rules, i.e., rules that produce new equi-valid sequents whose antecedent and consequent have shorter spatial parts. E.g., it is the case that

$$C * x \mapsto y \models D * x \mapsto y \iff C \models D.$$

The procedure terminates when an axiomatically valid sequent is produced, e.g., $\mathtt{emp} \models \mathtt{emp}$ or $C \models \mathtt{true}$.

The completeness of the algorithm rests on the fact that for a valid sequent $A \models \mathtt{ls}(x_1, y_1) * \ldots * \mathtt{ls}(x_n, y_n) * T$ (where T is `emp` or `true`) we can uniquely partition A into n sets A_i that form non-empty paths from x_i to y_i. □

Next we establish that the solutions to abduction can be obtained using variables only appearing in the antecedent and consequent. This, together with the fact that in consistent formulae there are *no repetitions of L-values* in different $*$-conjuncts (thus the formula $x \mapsto y \, * \, x \mapsto y'$ is inconsistent), allows us to conclude that only *polynomial-sized* candidate solutions need be considered.

Theorem 4.4 (Interpolation Theorem). *Let X be a solution to the abduction problem: $A * X \models B$. Then there is an $\widehat{X}$, a solution to the same abduction problem, such that $\widehat{X}$ uses only variables and constants mentioned in A or in B, and the spatial part of $\widehat{X}$ consists only of 'points-to' subformulae.*

Proof Sketch. We sketch the main ideas with an example X.

First, note that we can get rid of all `ls`-subformulae in X, since X will be still a solution to the problem, even if we replace each $\mathtt{ls}(x, y)$ occurring in X with the formula $(x{\neq}y) \wedge x \mapsto y$.

Suppose X of the form $X' * x_1 \mapsto z * \cdots * x_k \mapsto z * z \mapsto y$ is a solution to the abduction problem $A * X \models B$, and z does not occur in A, or B, or X'. In order to eliminate such an extra z, we replace X with $\widehat{X}$

$$\widehat{X} = X' * \, x_1 \mapsto y \, * \cdots * \, x_k \mapsto y.$$

To check that such an $\widehat{X}$ is a solution - that is, $(s, h) \models A * \widehat{X}$ implies $(s, h) \models B$, we construct a specific model (s', h_z) so that $(s', h_z) \models A * X$ with the original X.

Here we take advantage of the fact that z does not participate in h, and modify s with $s'(z) = c$ where c is *fresh*. To make h_z from h, we substitute the c for all occurrences of $s(y)$ in h related to $s(x_i)$ and then add the cell $c \rightarrow \boxed{\mathrm{s(y)}}$.

Being a solution, the original X provides that $(s', h_z) \models B$.

To complete the proof, it suffices to show that, because of our specific choice of the modified (s', h_z), the poorer (s, h) is a model for B as well. □

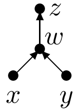
5 NP-completeness

We now show that the general separated abduction problem is NP-complete by reducing from 3-SAT.

The obstruction to a direct reduction from 3-SAT is that the Boolean disjunction $\vee$, which is a core ingredient of NP-hardness of the problem, is *not* expressible in our language. However, the following example illustrates how disjunctions over equalities and disequalities can be emulated through abduction.

Example 5.1. Define a formula A as follows, presented graphically to the right.

$$A \equiv x \mapsto w * y \mapsto w * w \mapsto z$$



Now, let X be an arbitrary solution to the abduction problem:

$$A * z \mapsto z' * X? \vDash u \neq v \wedge z \mapsto z' * \mathtt{ls}(x,u) * \mathtt{ls}(y,v) * \mathtt{true}$$

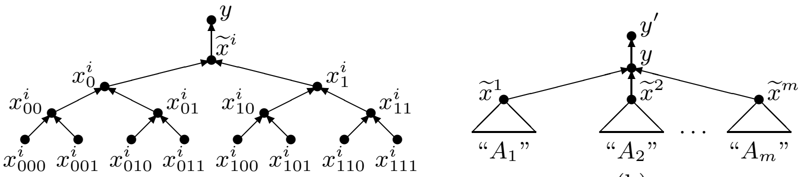
Then we can prove the following disjunction:

$$A * z \mapsto z' * X \vDash ((u = z) \wedge (v = w)) \vee ((v = z) \wedge (u = w)).$$

Thus any solution X provides either $\mathtt{ls}(x,z) * \mathtt{ls}(y,w)$, i.e. the path from x to z and the path from y to w, or $\mathtt{ls}(y,z) * \mathtt{ls}(x,w)$, i.e. the path from the leaf y to the root z and the path from the leaf x to the non-terminal vertex w. □

This mechanism, which utilizes the semantics of lists and separation in tandem, achieves emulation of disjunction over pure formulae. We generalize this in the combinatorial lemma 5.2 and then put it to use in our reduction from 3-SAT.

Lemma 5.2. *The tree in Fig. 2(a) has exactly eight non-overlapping (having no common edges) paths from its leaves to distinct non-terminal vertices. The disjunction we will emulate is provided by the fact that each path leading from a leaf to the root is realizable within such a partition into non-overlapping paths.*



(a) (b)

Fig. 2. (a) The graph presents A_i associated with a given clause C_i. (b) The graph presents the "whole" $A_0 * A_1 * A_2 * \cdots A_m$.

Now we can state our reduction. In essence, for each clause C_i we use a tree such as the one in Fig. 2a, and add appropriate disequalities so that any solution to the abduction problem selects a propositional valuation that satisfies all clauses.

Definition 5.3 (reduction). Given a set of 3-SAT clauses C_1,C_2,...,C_m, the problem we consider is to find an X, a solution to the abduction problem

$$A_0 * A_1 * \cdots * A_m * X \vDash \mathit{\Pi} \wedge A_0 * B_1 * \cdots * B_m * \mathtt{true} \qquad \text{(P1)}$$

where A_0 is $y \mapsto y'$, and each A_i is defined as a $*$-conjunction of all formulae of the form (see Fig. 2(a)):

$$x^i_{\varepsilon_1\varepsilon_2\varepsilon_3} \mapsto x^i_{\varepsilon_1\varepsilon_2}, \quad x^i_{\varepsilon_1\varepsilon_2} \mapsto x^i_{\varepsilon_1}, \quad x^i_{\varepsilon_1} \mapsto \widetilde{x}^i, \quad \widetilde{x}^i \mapsto y$$

where ε_1, ε_2, ε_3 range over zeros and ones, and B_i is defined as a $*$-conjunction of the form:

$$\mathtt{ls}(x^i_{000}, z^i_{000}) * \cdots * \mathtt{ls}(x^i_{\varepsilon_1\varepsilon_2\varepsilon_3}, z^i_{\varepsilon_1\varepsilon_2\varepsilon_3}) * \cdots * \mathtt{ls}(x^i_{111}, z^i_{111})$$

For each i, the non-spatial part $\mathit{\Pi}$ includes disequalities:

$$\begin{array}{ll} (z^i_{\varepsilon_1\varepsilon_2\varepsilon_3} \neq x^i_{\varepsilon_1\varepsilon_2\varepsilon_3}), & \\ (z^i_{\varepsilon_1\varepsilon_2\varepsilon_3} \neq z^i_{\delta_1\delta_2\delta_3}), & \text{for } (\varepsilon_1,\varepsilon_2,\varepsilon_3) \neq (\delta_1,\delta_2,\delta_3), \\ (z^i_{\varepsilon_1\varepsilon_2\varepsilon_3} \neq y), & \text{if } C_i(\varepsilon_1,\varepsilon_2,\varepsilon_3) \text{ is false.} \end{array} \qquad (1)$$

For distinct i and j, $\mathit{\Pi}$ also includes disequalities of the form

$$(z^i_{\varepsilon_1\varepsilon_2\varepsilon_3} \neq z^j_{\delta_1\delta_2\delta_3}) \qquad (2)$$

whenever $(\varepsilon_1,\varepsilon_2,\varepsilon_3)$ and $(\delta_1,\delta_2,\delta_3)$ are *incompatible* - that is, they assign contradictory Boolean values to a common variable u from C_i and C_j. □

5.1 From 3-SAT to the Abduction Problem (P1)

Let $(\alpha_1,\alpha_2,\ldots,\alpha_n)$ be an assignment of a value 0(false) or 1(true) to each of the Boolean variables such that it makes all clauses C_1,...,C_m true. Then we can find a solution $\widetilde{X}$ to our abduction problem in the following way.

By $(\beta^i_1,\beta^i_2,\beta^i_3)$ we denote the part of the assignment related to the variables used in C_i, so that $C_i(\beta^i_1,\beta^i_2,\beta^i_3)$ is true, and for all i and j, $(\beta^i_1,\beta^i_2,\beta^i_3)$ and $(\beta^j_1,\beta^j_2,\beta^j_3)$ are compatible.

Let v^i_1, v^i_2, v^i_3, v^i_4, v^i_5, v^i_6, v^i_7, v^i_8 denote y, $\widetilde{x}^i$, x^i_0, x^i_1, x^i_{00}, x^i_{01}, x^i_{10}, x^i_{11}, the non-terminal vertices in Fig. 2. As in Lemma 5.2, we construct eight non-overlapping paths leading from x^i_{000}, x^i_{001},..., x^i_{111} to distinct $v^i_{k_1}$, $v^i_{k_2}$,..., $v^i_{k_8}$, respectively, so that one path leads from $x^i_{\beta^i_1\beta^i_2\beta^i_3}$ to v^i_1, where $(\beta^i_1,\beta^i_2,\beta^i_3)$ is specified above. The part X_i is defined as a set of the following equalities

$$\begin{array}{l} (z^i_{000} = v^i_{k_1}),\ (z^i_{001} = v^i_{k_2}),\ (z^i_{010} = v^i_{k_3}),\ (z^i_{011} = v^i_{k_4}), \\ (z^i_{100} = v^i_{k_5}),\ (z^i_{101} = v^i_{k_6}),\ (z^i_{110} = v^i_{k_7}),\ (z^i_{111} = v^i_{k_8}) \end{array}$$

which contains, in particular, the equality $(z^i_{\beta^i_1\beta^i_2\beta^i_3} = y)$.

Example 5.4. Fig. 3 yields the following X_i:

$$\begin{array}{l} (z^i_{000} = x^i_{00}),\ (z^i_{001} = x^i_0),\ (z^i_{010} = x^i_{01}),\ (z^i_{011} = y), \\ (z^i_{100} = x^i_{10}),\ (z^i_{101} = x^i_1),\ (z^i_{110} = x^i_{11}),\ (z^i_{111} = \widetilde{x}^i) \end{array}$$

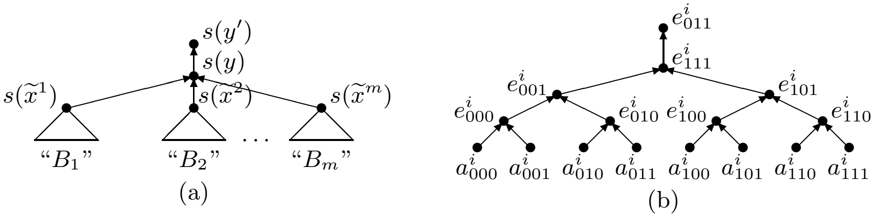


Fig. 3. (a) The graph depicts (s, h), a model for $A_0 * B_1 * B_2 * \cdots B_m$. (b) The graph depicts the part (s, h_i) such that $(s, h_i) \vDash B_i$. Here, the following properties hold, $a^i_{\varepsilon_1\varepsilon_2\varepsilon_3} = s(x^i_{\varepsilon_1\varepsilon_2\varepsilon_3})$, and $e^i_{\varepsilon_1\varepsilon_2\varepsilon_3} = s(z^i_{\varepsilon_1\varepsilon_2\varepsilon_3})$, and $s(z^i_{011}) = s(y)$.

Lemma 5.5. *With* $\widetilde{X} = \Pi \wedge X_1 \wedge X_2 \wedge \cdots \wedge X_m$ *we get a solution to (the non-relaxed version of) the abduction problem (P1):*

$$\widetilde{X}? \wedge A_0 * A_1 * A_2 * \cdots A_m \vDash \Pi \wedge A_0 * B_1 * B_2 * \cdots B_m$$

Proof. It suffices to show that $X_i \wedge A_i \vDash B_i$ is valid for each i. □

5.2 From the Abduction Problem (P1) to 3-SAT

Here we prove that our encoding is faithful.

Lemma 5.6. *Given an* X*, a solution to the abduction problem (P1), we can construct an assignment* $(\alpha_1, \alpha_2, \ldots, \alpha_n)$ *that makes all clauses* $C_1, \ldots, C_m$ *true.*

Proof. Assume $(s, h') \vDash A_0 * A_1 * A_2 * \cdots A_m * X$.
Then $(s, h) \vDash A_0 * B_1 * B_2 * \cdots B_m$ for a 'sub-heap' h, and h can be split in heaps $\widehat{h}$ and h_1, $h_2, \ldots, h_m$, so that $(s, \widehat{h}) \vDash y \mapsto y'$, and

$$(s, h_1) \vDash B_1,\ (s, h_2) \vDash B_2, \ldots,\ (s, h_m) \vDash B_m,$$

respectively. The non-overlapping conditions provide that any path in each of the h_i is blocked by the *'bottleneck'* A_0 and hence cannot go beyond $s(y)$. Therefore, the whole h must be of the form shown in Fig. 3(a), and each of the h_i must be of the form shown in Fig. 3(b).

For every B_i, to comply with Π, these eight values $s(z^i_{\varepsilon_1\varepsilon_2\varepsilon_3})$ must be one-to-one assigned to the eight non-terminal vertices in the tree in Fig. 3(b) (see Lemma 5.2).

Hence for each of the "non-terminal" variables v^i_1, $v^i_2, \ldots$, v^i_8 in Fig. 2(a), X must impose equalities of the form

$$s(z^i_{\varepsilon_1\varepsilon_2\varepsilon_3}) = s(v^i_{k_\ell}).$$

In particular, $s(z^i_{\delta^i_1\delta^i_2\delta^i_3}) = s(y)$ for some $(\delta^i_1, \delta^i_2, \delta^i_3)$. To be consistent with the third line of (1), $C_i(\delta^i_1, \delta^i_2, \delta^i_3)$ must be true. In addition to that, for distinct i and j, we get

$$s(z^i_{\delta^i_1\delta^i_2\delta^i_3}) = s(y) = s(z^j_{\delta^j_1\delta^j_2\delta^j_3}),$$

which makes these $(\delta^i_1, \delta^i_2, \delta^i_3)$ and $(\delta^j_1, \delta^j_2, \delta^j_3)$ compatible, in accordance with (2).

We assemble the desired assignment $(\alpha_1, \alpha_2, \ldots, \alpha_n)$ in the following way.

For any Boolean variable u occurring in some clause C_i we take the triple $(\delta_1^i, \delta_2^i, \delta_3^i)$ specified above and assign the corresponding δ_ℓ^i to u. The fact that all $(\delta_1^i, \delta_2^i, \delta_3^i)$ and $(\delta_1^j, \delta_2^j, \delta_3^j)$ are compatible provides that our assignment procedure is well-defined. □

Theorem 5.7. *The following problems are* NP*-complete, given formulae A, B:*

(a) *Determine if there is a solution X to the problem $A * X? \models B$.*

(b) *Determine if there is a solution X to the problem $A * X? \models B * \mathtt{true}$.*

(c) *Determine if there is a solution X to the problem $A * X? \models B * \mathtt{true}$ in the case where the spatial part of A uses only $\mapsto$-subformulae, and the models are confined to the heaps the length of any acyclic path in which is bounded by* 5 *(even if the spatial part of X is supposed to be the trivial* $\mathtt{emp}$*).*

Proof. It follows from Lemmas 5.5 and 5.6, Theorem 4.1, and the fact that the tree height in Fig. 3(a) is bounded by 5. □

Remark 5.8. It might seem that NP-hardness for rSAP($\mapsto$, $\mathtt{ls}$) should necessarily use lists of unbounded length. But, our encoding exploits only list segments of length no more than 5.

Remark 5.9. By Theorem 5.7, any solving algorithm running in polynomial time is likely to be *incomplete*. Consider, for instance, the abduction problem

$$x \mapsto y * y \mapsto z * w \mapsto y * X? \models \mathtt{ls}(x, a) * \mathtt{ls}(w, a) * \mathtt{true}.$$

There is a solution, namely, $y = a \wedge \mathtt{emp}$. However, the polynomial-time algorithm presented in [6] would stop without producing a solution, hence the incompleteness of that algorithm.

6 NP-completeness and PTIME Results for $\mapsto$ Fragments

We have seen that the abduction problem for symbolic heaps is NP-complete in general. In this section we consider a restricted collection of formulae, which contain $\mapsto$ but not $\mathtt{ls}$. Such formulae occur often in program analysis, and form an important sub-case of the general problem. Here we find a perhaps surprising phenomenon: the general problem SAP($\mapsto$) is NP-complete, but the 'relaxed' problem rSAP($\mapsto$) can be solved in polynomial time. The relaxed problem, which has $*\mathtt{true}$ in the consequent, is relevant to program analysis (and is used in the Abductor tool), and thus the polynomial case is of practical importance.

6.1 SAP($\mapsto$) is NP-complete

Here, we show NP-hardness of SAP($\mapsto$) by reducing from the 3-partition problem [15]. The intuitions behind this reduction are as follows. (a) We coerce the abduction solution, if it exists, to be a conjunction of equalities, with $\mathtt{emp}$ as the spatial part; here the absence of $\mathtt{true}$ in the consequent is crucial. (b) The separating conjunction enables us to specify that distinct parts of the antecedent must be matched, via equalities, to distinct parts of the consequent.

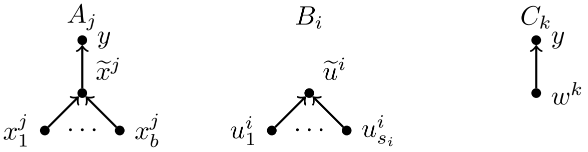


Fig. 4. The formulae used in Definition 6.1

Definition 6.1 (reduction). Given the 3-*partition problem*:

> Given an integer bound b and a set of $3m$ integers s_1, s_2,..., s_{3m}, strictly between $b/4$ and $b/2$, decide if these numbers can be partitioned into triples $(s_{i_1}, s_{i_2}, s_{i_3})$ so that $s_{i_1}+s_{i_2}+s_{i_3} = b$.

the problem we consider is to find an X, a solution to the abduction problem

$$A_1 * \cdots * A_m * X? \vDash \mathit{\Pi} \wedge B_1 * \cdots * B_{3m} * C_1 * \cdots * C_m \tag{P2}$$

where A_j, B_i, C_k, and $\mathit{\Pi}$ are defined as follows, here j and k range from 1 to m, and i ranges from 1 to $3m$ (cf. Fig. 6.1):

$A_j \;=\; x_1^j \mapsto \widetilde{x}^j \;*\; x_2^j \mapsto \widetilde{x}^j \;* \cdots *\; x_b^j \mapsto \widetilde{x}^j \;*\; \widetilde{x}^j \mapsto y$
$B_i \;=\; u_1^i \mapsto \widetilde{u}^i \;*\; u_2^i \mapsto \widetilde{u}^i \;* \cdots *\; u_{s_i}^i \mapsto \widetilde{u}^i$
$C_k \;=\; w^k \mapsto y$, and $\mathit{\Pi}$ consists of all disequalities of the form $y \neq x_\ell^j$, $y \neq \widetilde{x}^j$, $\widetilde{u}^i \neq x_\ell^j$, $\widetilde{u}^i \neq y$, and $w^k \neq x_\ell^j$. □

6.1.1 From 3-partition to the Abduction Problem (P2)

Here we transform any solution to the 3-partition problem into a solution $\widetilde{X}$ to the abduction problem (P2).

Suppose that for any j, $s_{3j-2}+s_{3j-1}+s_{3j} = b$.
First, we take $\mathit{\Pi}$ as $\widetilde{X}$ and add to it all equalities of the form

$$\widetilde{u}^{3j-2} = \widetilde{u}^{3j-1} = \widetilde{u}^{3j} = \widetilde{x}^j = w^j.$$

For each j, we include in $\widetilde{X}$ all equalities of the form

$$(x_1^j = u_1^{3j-2}),\ \ldots,\ (x_\ell^j = t_j(x_\ell^j)),\ \ldots,\ (x_b^j = u_{s_{3j}}^{3j})$$

where t_j is a bijection t_j between the sets $\{x_1^j, .., x_b^j\}$
and $\{u_1^{3j-2}, .., u_{s_{3j-2}}^{3j-2}, u_1^{3j-1}, .., u_{s_{3j-1}}^{3j-1}, u_1^{3j}, .., u_{s_{3j}}^{3j}\}$. □

Lemma 6.2. *$\widetilde{X} \wedge A_j \vDash B_{3j-2} * B_{3j-1} * B_{3j} * C_j$ is valid for every j. Hence $\widetilde{X}$ is a solution to the problem (P2).*

6.1.2 From the Abduction Problem (P2) to 3-partition

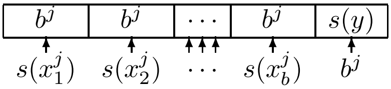
Here we prove that our encoding is faithful.

Lemma 6.3. *Given an X, a solution to (P2), we can construct a solution to the related 3-partition problem.*

Proof. We suppose that $\sum_{i=1}^{3m} s_i = mb$.

Assume $(s,h) \vDash A_1 * A_2 * \cdots A_m * X$.

Then h can be split in heaps h_1, h_2,…, h_m, and h', so that $(s,h_1) \vDash A_1$, $(s,h_2) \vDash A_2$,…, $(s,h_m) \vDash A_m$, and $(s,h') \vDash X$, respectively.
More precisely, each h_j is of the form



where $b^j = s(\widetilde{x}^j)$. Notice that h_j can be uniquely identified by b^j.

Because of (P2), $(s,h) \vDash B_1 * \cdots * B_{3m} * C_1 * \cdots C_m$.

The left-hand side of (P2) indicates the size of h as $m(b+1)$ plus the size of h'. The right-hand side of (P2) shows that the size of h is exactly $m(b+1)$. Bringing all together, we conclude that h' is the empty heap.

Let f_i be a part of h such that $(s,f_i) \vDash C_i$, and g_i be a part of h such that $(s,g_i) \vDash B_i$. To comply with Π, every $s(w^k)$ must be one-to-one assigned to one of these m points b^1,…, b^m, and every $s(\widetilde{u}^i)$ must be assigned to one of these b^1,…, b^m, as well. The effect is that each h_j is the exact composition of a one-cell heap $f_{i'}$ and heaps g_{j_1}, g_{j_2},…, g_{j_ℓ}, whose domains are disjoint, and hence $1+s_{j_1}+s_{j_2}+\cdots+s_{j_\ell} = b+1$, which provides that $\ell = 3$ and thereby the desired instance of the 3-partition problem. □

Theorem 6.4. *The following problems are* NP*-complete:*

(a) *Given formulae A and B whose spatial parts use only $\mapsto$-subformulae, determine if there is a solution X to the abduction problem $A * X? \vDash B$.*

(b) *Given formulae A and B whose spatial parts use only $\mapsto$-subformulae, determine if there is a solution X to the abduction problem $A * X? \vDash B$ in the case where the models are confined to the heaps the length of any acyclic path in which is bounded by 2 (even if the spatial part of X is supposed to be* `emp`*).*

Proof. It follows from Lemmas 6.2 and 6.3 and Theorem 4.1, and the fact that the longest path in h_j is of length 2. □

6.2 rSAP($\mapsto$) Is in PTIME

We will assume that all formulae in this subsection contain no list predicates, and make no more mention of this fact. We will also use the notation alloc(A) to denote the set of variables x such that there is an L-value y in A and $A \vDash x = y$.

An algorithm which constructs a solution for $A * X? \vDash B * \texttt{true}$ if one exists or fails if there is no solution, is as follows. We check the consistency of the pure consequences of A and B and return false if they are not consistent. Then we transform the problem into one of the form $A' * X? \vDash \Sigma * \texttt{true}$, i.e., the consequent has no pure part, while guaranteeing that the two problem are in a precise sense equivalent. Next we subtract $\mapsto$-predicates from A' and Σ that have the same L-value. This step may also generate requirements (equalities) that the solution must entail. Finally we show that if this step cannot be repeated any more, then $A' * \Sigma$ is consistent and therefore Σ is a solution. We then transform this solution back to one for the original abduction problem.

Lemma 6.5. *The abduction problem $A * X? \models \Pi \wedge \Sigma$ has a solution if and only if $(\Pi \wedge A) * X? \models \Sigma$ has a solution. Moreover, if X is a solution for $(\Pi \wedge A) * X? \models \Sigma$, then $\Pi \wedge X$ is a solution for $A * X? \models \Pi \wedge \Sigma$.*

Thus we can concentrate on instances where the consequent has no pure part.

Lemma 6.6. *The following conjunctions are equivalent:*

$$\left\{\begin{array}{l} A \textit{ consistent} \\ A \models x = y \wedge w = z \wedge B \\ x \notin \mathrm{alloc}(A) \end{array}\right\} \iff \left\{\begin{array}{l} A \models x = y \\ A * x \mapsto w \textit{ consistent} \\ A * x \mapsto w \models B * y \mapsto z \end{array}\right\}$$

Proof. Left-to-right: The entailment $A * x \mapsto w \models B * y \mapsto z$ follows by $*$-introduction. The consistency of $A * x \mapsto w$ follows easily by an argument that whenever $x \notin \mathrm{alloc}(C)$ for some consistent C, there exists a model (s, h) of C that does not include $s(x)$ in the domain of h.

Right-to-left: since $A * x \mapsto w$ is consistent then so is A. It is also easy to see that if $A \not\models w = z$ then $A \wedge w \neq z$ is consistent, thus there exist a countermodel for $A * x \mapsto w \models B * y \mapsto z$ where the cell at address $s(x)$ contains a value $c \neq s(z)$. Also, from the assumption of consistency trivially follows that $x \notin \mathrm{alloc}(A)$.

Let $A \equiv E \wedge A'$ where E are the equalities appearing in A. Then,

$$A * x \mapsto w \models B * y \mapsto z \implies A'[E] * (x \mapsto w)[E] \models B[E] * (y \mapsto z)[E]$$

where the $[E]$ notation indicates substitution through the equalities in E. Let $a \equiv x[E]$ and $b \equiv w[E]$. Then, we can derive

$$A'[E] * a \mapsto b \models B[E] * a \mapsto b.$$

Let $A'' * a \mapsto b$ be the explicit, equivalent, version of $A'[E] * a \mapsto b$. Then it can be shown that $A'' * a \mapsto b \models B[E] * a \mapsto b$ implies $A'' \models B[E]$. Thus, $E \wedge A'' \models E \wedge B[E]$ and, by the substitution rule, $A \models B$. □

Lemma 6.7. *Suppose A and Σ are such that (a) there are variables x,y such than $x \in \mathrm{alloc}(A)$, $y \in \mathrm{alloc}(\Sigma)$ and $A \models x = w$, and (b) there are no distinct predicates $x \mapsto y$, $w \mapsto z$ in Σ for which $A \models x = w$. Then $A * \Sigma$ is consistent.*

Theorem 6.8 (rSAP($\mapsto$) **is in** PTIME)**.** *There is a polytime algorithm that finds a solution for $A * X? \models B *$ `true`, or answers no otherwise.*

Proof. Let $B \equiv \Pi \wedge \Sigma$. If $\Pi \wedge A$ is inconsistent then clearly there is no solution, and we can check this in polynomial time.

The abduction problem $(\Pi \wedge A) * X? \models \Sigma *$ `true` has a solution iff the original one has and we know how to transform a solution of the former to one of the latter through Lemma 6.5. Thus from now on we solve $(\Pi \wedge A) * X? \models \Sigma *$ `true`.

We repeatedly subtract the $\mapsto$ predicates from antecedent and consequent for which the rule in Lemma 6.6 applies, to obtain an abduction problem with the side condition that certain variables may not appear as L-values in the solution.

When this is no longer possible we check the antecedent for consistency, as it is possible that the equalities introduced through this step contradict some disequality. For example, $y \neq w \wedge x \mapsto y * X? \vDash x \mapsto w * \mathtt{true}$.

Ultimately, we will arrive at an abduction problem $(\Pi' \wedge \Sigma') * X? \vDash \Sigma'' * \mathtt{true}$ which satisfies the conditions of Lemma 6.7. In this case, Σ'' is a solution as $(\Pi' \wedge \Sigma') * \Sigma''$ is consistent, and trivially, $(\Pi' \wedge \Sigma') * \Sigma'' \vDash \Sigma'' * \mathtt{true}$.

Thus we return $\Pi' \wedge \Sigma''$ as the solution for the original problem. □

Remark 6.9. This result may seem surprising as rSAP($\mapsto$, $\mathtt{ls}$) and SAP($\mapsto$) are both NP-complete.

To illustrate the differences between rSAP($\mapsto$) and rSAP($\mapsto$, $\mathtt{ls}$) that allow this divergence, we consider the two abduction problems.

$$A * x \mapsto a_1 * a_1 \mapsto a_2 * \cdots * a_{k-1} \mapsto a_k * X? \vDash x \mapsto y * B * \mathtt{true} \tag{3}$$

$$A * x \mapsto a_1 * a_1 \mapsto a_2 * \cdots * a_{k-1} \mapsto a_k * X? \vDash \mathtt{ls}(x, y) * B * \mathtt{true} \tag{4}$$

One of the ingredients that makes the algorithm in Theorem 6.8 polynomial-time is the fact that for any solution X to the abduction problem (3), there is no other choice but to take y as a_1. On the other hand, in problem (4), y can be made equal to a_1, to a_2,..., to a_k, or something else. *It is this phenomenon we exploit (even with $k \leq 5$) to achieve* NP*-hardness in Theorem 5.7*.

In the case of rSAP($\mapsto$) and SAP($\mapsto$) another factor is at play. Consider, e.g., the two abduction problems, where $x_1,\ldots,x_k$ are not L-values in the LHS:

$$A * a_1 \mapsto b_1 * \ldots * a_k \mapsto b_k * X? \vDash B * x_1 \mapsto y_1 * \ldots * x_k \mapsto y_k \tag{5}$$

$$A * a_1 \mapsto b_1 * \ldots * a_k \mapsto b_k * X? \vDash B * x_1 \mapsto y_1 * \ldots * x_k \mapsto y_k * \mathtt{true} \tag{6}$$

To find a solution to (5), because of precision (Remark 2.3), $x_1,\ldots,x_k$ must be assigned to L-values in the left-hand side. We have at least $k!$ possibilities and each may need to be checked (*this point lies behind the* NP*-hardness in Theorem 6.4*).

In contrast, to find a solution to (6) we can opt for the most "conservative" candidate solution leaving $x_1,\ldots,x_k$ unassigned, or in other words, we can include $x_1 \mapsto y_1 * \ldots * x_k \mapsto y_k$ as a part of a candidate solution X, since

$$a_1 \mapsto b_1 * \ldots * a_k \mapsto b_k * X \vDash x_1 \mapsto y_1 * \ldots * x_k \mapsto y_k * \mathtt{true}.$$

If such an X is *not* a solution then problem (6) has no solution at all.

These two observations are among the crucial points behind the design of the polynomial-time algorithm in Theorem 6.8. □

7 Conclusion

Our results are summarized in the table below.

	SAP	rSAP
$\{\mapsto\}$	NP-complete	in PTIME
$\{\mapsto, \mathtt{ls}\}$	NP-complete	NP-complete

We have studied the complexity of abduction for certain logical formulae representative of those used in program analysis for the heap. Naturally, practical program analysis tools (e.g., Abductor, SLAyer, Infer, Xisa) use more complicated predicates in order to be able to deal with examples arising in practice. For example, to analyze a particular device driver a formula was needed corresponding to 'five cyclic linked lists sharing a common header node, where three of the cyclic lists have nested acyclic sublists' [1].

The fragment we consider is a basic core used in program analysis. All of the separation logic-based analyses use at least a points-to and a list segment predicate. So our lower bounds likely carry over to richer languages used in tools.

Furthermore, the work here could have practical applications. The tools use abduction procedures that are incomplete but the ideas here can be used to immediately obtain procedures that are less incomplete (even when the fragment of logic they are using is not known to be decidable). Further, the polynomial-time sub-cases we identified correspond to cases that do frequently arise in practice. For example, when the Abductor tool was run on an IMAP server of around 230k LOC [11], it found consistent pre/post specs for 1150 out of 1654 procedures, and only 37 (or around 3%) of the successfully analyzed procedures had specifications involving list segments. The remainder (97% of the successfully analyzed procedures, or 67% of all procedures) had specs lying within the $\mapsto$-fragment which has a polynomial-time relaxed abduction problem (and the tool uses the relaxed problem). Furthermore, the 37 specifications involving list segments did not include any assertions with more than one list predicate. Indeed, we might hypothesize that in real-world programs one would only rarely encounter a single procedure that traverses a large number of distinct data structures, and having a variable number of list segment predicates was crucial in our NP-hardness argument. Because of these considerations, we expect that the worst cases of the separated abduction problem can often be avoided in practice.

Acknowledgements. This research was suppored by funding from the EPSRC. O'Hearn also acknowledges the support of a Royal Society Wolfson Research Merit award and a gift from Microsoft Research.

References

1. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W., Wies, T., Yang, H.: Shape analysis for composite data structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 178–192. Springer, Heidelberg (2007)
2. Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic execution with separation logic. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 52–68. Springer, Heidelberg (2005)
3. Bjørner, N., Hendrix, J.: Linear functional fixed-points. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 124–139. Springer, Heidelberg (2009)
4. Brotherston, J., Kanovich, M.I.: Undecidability of propositional separation logic and its neighbours. In: LICS, pp. 130–139. IEEE Computer Society, Los Alamitos (2010)
5. Calcagno, C., Distefano, D.: Infer: an automatic program veriifier for memory safety of C programs. In: To appear in 3rd NASA Formal Methods Symposium (2011)

6. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. In: 36th POPL, pp. 289–300 (2009)
7. Calcagno, C., Distefano, D., Vafeiadis, V.: Bi-abductive resource invariant synthesis. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 259–274. Springer, Heidelberg (2009)
8. Calcagno, C., O'Hearn, P.W., Yang, H.: Local action and abstract separation logic. In: LICS (2007)
9. Cousot, P., Cousot, R.: Modular static program analysis. In: CC 2002. LNCS, vol. 2304, pp. 159–178. Springer, Heidelberg (2002)
10. Creignou, N., Zanuttini, B.: A complete classification of the complexity of propositional abduction. SIAM J. Comput. 36(1), 207–229 (2006)
11. Distefano, D.: Attacking large industrial code with bi-abductive inference. In: Alpuente, M., Cook, B., Joubert, C. (eds.) FMICS 2009. LNCS, vol. 5825, pp. 1–8. Springer, Heidelberg (2009)
12. Distefano, D., Filipović, I.: Memory leaks detection in java by bi-abductive inference. In: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 278–292. Springer, Heidelberg (2010)
13. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: Hermanns, H. (ed.) TACAS 2006. LNCS, vol. 3920, pp. 287–302. Springer, Heidelberg (2006)
14. Eiter, T., Gottlob, G.: The complexity of logic-based abduction. J. ACM 42(1), 3–42 (1995)
15. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman, New York (1979)
16. Giacobazzi, R.: Abductive analysis of modular logic programs. In: Proc. of the 1994 International Logic Prog. Symp., pp. 377–392. The MIT Press, Cambridge (1994)
17. Gulavani, B., Chakraborty, S., Ramalingam, G., Nori, A.: Bottom-up shape analysis. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 188–204. Springer, Heidelberg (2009)
18. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: 35th POPL, pp. 235–246 (2008)
19. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W.: Scalable shape analysis for systems code. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 385–398. Springer, Heidelberg (2008)
20. Ishtiaq, S., O'Hearn, P.W.: BI as an assertion language for mutable data structures. In: Proceedings of the 28th POPL, pp. 14–26 (2001)
21. Lahiri, S.K., Qadeer, S.: Back to the future: revisiting precise program verification using SMT solvers. In: 35th POPL, pp. 171–182 (2008)
22. Luo, C., Craciun, F., Qin, S., He, G., Chin, W.-N.: Verifying pointer safety for programs with unknown calls. Journal of Symbolic Computation 45(11), 1163–1183 (2010)
23. Möller, A., Schwartzbach, M.I.: The pointer assertion logic engine. In: 22nd PLDI, pp. 221–231 (2001)
24. Paul, G.: Approaches to abductive reasoning: an overview. Artif. Intell. Rev. 7(2), 109–152 (1993)
25. Peirce, C.S.: The collected papers of Charles Sanders Peirce. Harvard University Press, Cambridge (1958)
26. Sagiv, M., Reps, T., Wilhelm, R.: Solving shape-analysis problems in languages with destructive updating. ACM TOPLAS 20(1), 1–50 (1998)
27. Yorsh, G., Rabinovich, A.M., Sagiv, M., Meyer, A., Bouajjani, A.: A logic of reachable patterns in linked data-structures. J. Log. Algebr. Program. 73(1-2), 111–142 (2007)

Efficient Decision Procedures for Heaps Using STRAND

P. Madhusudan and Xiaokang Qiu

University of Illinois at Urbana-Champaign, USA
{madhu,qiu2}@illinois.edu

Abstract. The STRAND [10] logic allows expressing structural properties of heaps combined with the data stored in the nodes of the heap. A semantic fragment of STRAND as well as a syntactically defined subfragment of it are known to be decidable [10]. The known decision procedure works by combining a decision procedure for MSO on trees (implemented by the tool MONA) and a decision procedure for the quantifier-free fragment of the data-theory (say, integers, and implemented using a solver like Z3).

The known algorithm for deciding the syntactically defined decidable fragment (which is the same as the one for the semantically defined decidable fragment) involves solving large MSO formulas over trees, whose solution is the main bottleneck in obtaining efficient algorithms. In this paper, we focus on the syntactically defined decidable fragment of STRAND, and obtain a new and more efficient algorithm. Using a set of experiments obtained from verification conditions of heap-manipulating programs, we show the practical benefits of the new algorithm.

1 Introduction

Several approaches to program analysis, like deductive verification, generating tests using constraint solving, abstraction, etc. have greatly benefited from the engineering of efficient SMT solvers, which currently provide automated decision procedures for a variety of quantifier-free theories, including integers, bit-vectors, arrays, uninterpreted functions, as well as *combinations* of these theories using the Nelson-Oppen method [15]. One of the most important kinds of reasoning in program verification that has evaded tractable decidable fragments is reasoning with dynamic heaps and the data contained in them.

Reasoning with heaps and data seems to call for decidable combinations of logics on *graphs* that model the heap structure (with heap nodes modeled as vertices, and field pointers as edges) with a logic on the data contained in them (like the quantifier-free theory of integers already supported by current SMT solvers). The primary challenge in building such decidable combinations stems from the *unbounded* number of nodes in the heap structures. This mandates the need for universal quantification in any reasonable logic in order to be able to refer to all the elements of the heap (e.g. to say a list is sorted, we need some form of universal quantification). However, the presence of quantifiers immediately

E. Yahav (Ed.): SAS 2011, LNCS 6887, pp. 43–59, 2011.

annuls the use of Nelson-Oppen combinations, and requires a new theory for combining unbounded graph theories with data.

Recently, we have introduced, along with Gennaro Parlato, a new logic called Strand that combines heap structures and data [10]. We also identified a *semantically* defined decidable fragment of Strand, called $\text{Strand}_{dec}^{sem}$, as well as a syntactic decidable fragment, called Strand_{dec}. The decision procedures for satisfiability for both the semantic and syntactic fragments were the same, and were based on (a) abstracting the data-predicates in the formula with Boolean variables to obtain a formula purely on graphs, (b) extracting the set of *minimal graphs* according to a satisfiability-preserving embedding relation that was completely agnostic to the data-logic, and is guaranteed to be minimal for the two fragments, and (c) checking whether any of the minimal models admits a data-extension that satisfies the formula, using a data-logic solver. We also showed that the decidable fragments can be used in the verification of pointer-manipulating programs. We implemented the decision procedure using Mona on tree-like data-structures for the graph logic and Z3 for quantifier-free arithmetic, and reported experimental results in Hoare-style deductive verification of certain programs manipulating data-structures.

In this paper, we concentrate on the *syntactic* decidable fragment of Strand identified in the previous work [10], and develop new efficient decision procedures for it. The bottleneck in the decision procedure of the current methods for deciding Strand is the phase that computes the set of all minimal models. This is done using monadic second-order logic (MSO) on trees, and is achieved by a *complex* MSO formula that has quantifier alternations and also includes adaptations of the Strand formula *twice* within it. In experiments, this phase is clearly the bottleneck (for example, a verification condition for binary search trees takes about 30s while the time spent by Z3 on the minimal models is less than 0.5s).

We propose in this paper a new method to solve satisfiability for Strand_{dec} using a notion called *small models*, which are not the precise set of minimal models but a slightly larger class of models. We show that the set of small models is always bounded, and also equisatisfiable (i.e. if there is any model that satisfies the Strand_{dec} formula, then there is a data-extension of a small model that satisfies it). The salient feature of small models is that it can be expressed by a much simpler MSO formula that is *completely independent of the* Strand_{dec} *formula*! The definition of small models depends only on the *signature* of the formula (in particular, the set of variables existentially quantified). Consequently, it does not mention any structural abstractions of the formula, and is much simpler to solve. This formulation of decidability is also a theoretical contribution, as it gives a much simpler alternative proof that the logic Strand_{dec} is decidable.

We implement the new decision procedure, and show, using the same set of experiments as in [10], that the new procedure's performance is at least an order-of magnitude faster than the known one (in some examples, the new algorithm works even 1000 times faster).

In summary, this paper builds a new decision procedure for STRAND$_{dec}$ that is based on new theoretical insights, and that is considerably faster than the known decision procedure. We emphasize that the new decision procedure, though faster, is sound and complete in deciding the logic STRAND$_{dec}$. In developing STRAND, we have often been worried on the reliance of automata-theoretic decision procedures (like MONA), which tend to perform badly in practice for large examples. However, the decision procedure for STRAND crucially uses the minimal model property that seems to require solving MSO formulas, which in turn are currently handled most efficiently by automata-theoretic tools. This paper shows how the automata-theoretic decision procedures can be used on much more simplified formulas in building fast decision procedures for heaps using STRAND.

The paper is structured as follows. In Section 2, we give a high-level overview of STRAND followed by definitions that we need in this paper, including the notions of recursively defined data-structures, the notion of submodels, the definition of elastic relations, and the decidable fragment of STRAND. We do not describe the known decision procedure for the syntactic decidable fragment of STRAND— we refer the reader to [10]. Section 3 describes the new theoretical decision procedure for STRAND$_{dec}$ and compares it, theoretically, with the previously known decision procedure. Section 4 describes experimental comparisons of the new decision procedure with the old.

A brief note on notation: In the sequel, we will refer to the general STRAND logic [10] as STRAND*, and refer to the syntactic decidable fragment as STRAND (which is called STRAND$_{dec}$ in [10]).

2 Data-Structures, Submodels, Elasticity and STRAND

In this section, we first give an informal overview of STRAND that will help understand the formal definitions. We then proceed to formally define the concepts of recursively defined data-structures, the notion of valid subsets of a tree, which in turn define submodels, define formulas that allow us to interpret an MSO formula on a submodel, and define the logic STRAND using elastic relations.

We refer the reader to [10] for more details, including the motivations for the design of the logic, how verification conditions can be formally extracted from code with pre- and post-conditions, as well as motivating examples.

2.1 An Overview of STRAND

We give first an informal overview of the logic STRAND*, the syntactic decidable fragment STRAND, and how they can be used for verifying heap-manipulating programs, as set forth in [10].

We model heap structures as labeled directed graphs: the nodes of the graph correspond to heap locations, and an edge from n to n' labeled f represents the fact that the field pointer f of node n points to n'. The nodes in addition have *labels* associated to them; labels are used to signify special nodes (like NIL nodes) as well as to denote the program's pointer variables that point to them.

Strand formulas are expressed over a particular class of heaps, called *recursively defined data-structures.* A (possibly infinite) set of recursively-defined data-structures is given by a tuple $(\psi_{Tr}, \psi_U, \{\alpha_a\}_{a \in L_v}, \{\beta_b\}_{b \in L_e})$. Here, ψ_{Tr} is an MSO formula on trees that defines a regular set of trees R, which forms the *backbone* skeletons of the structures, $\psi_U(x)$ is a monadic predicate expressed in MSO over trees that defines, for each tree, the subset of nodes that correspond to heap nodes, and the unary predicates α_a and binary predicates β_b, written in MSO over trees, identify the nodes labeled a and edges labeled b, respectively. The graphs that belong to the recursive data-structure are hence obtained by taking some tree T satisfying ψ_{Tr}, taking the subset of tree nodes of T that satisfy ψ_U as the nodes of the graph, and taking the a-labeled edges in the graph as those defined by E_a, for each $a \in \Sigma$.

The above way of defining graph-structures has many nice properties. First, it allows defining graphs in a way so that the MSO-theory on the graphs is decidable (by interpreting formulas on the underlying tree). Second, several naturally defined recursive data-structures in programs can be easily embedded in the above notation automatically. Intuitively, a recursive data-structure, such as a list of nodes pointing to trees or structs, has a natural skeleton which follows from the recursive data-type definition itself. In fact, *graph types* (introduced in [7]) are a simple textual description of recursive data-structures that are automatically translatable to our notation. Several structures including linked lists, doubly linked lists, cyclic and acyclic lists, trees, hierarchical combinations of these structures, etc., are definable using the above mechanism.

A Strand* formula is defined over a set of recursively-defined data-structures, and is of the form $\exists \vec{x} \forall \vec{y} \varphi(\vec{x}, \vec{y})$, where φ is a formula that combines MSO over graphs defined by a recursively-defined data structure, as well as logical constraints over the data stored in the heap nodes.

A *decidable* fragment of Strand*, called Strand (and called Strand$_{dec}$ in [10]), is defined over a signature consisting of *elastic* relations and *non-elastic* relations, and allows formulas of the kind $\exists \vec{x} \forall \vec{y} \varphi(\vec{x}, \vec{y})$ where φ has no further quantification, and where all non-elastic relations are restricted to the existentially quantified variables $\vec{x}$ (see below for formal definitions and see Fig. 2 for the syntax of the logic). Elastic relations have a technical definition: they are those that hold on any properly defined *sub-model* iff they hold in a model (see below for precise details). For example, on a tree, the *descendent* relation is an elastic relation, while the *child* relation is not.

In verifying programs, we require the user to give proof annotations that include pre-conditions and post-conditions and loop-invariants in Strand$^*_{\exists,\forall}$, the fragment of Strand* with a pure existential or universal prefix. The basic paths in the program hence become *assignments* and *assume* statements, and the invalidity of the Hoare-triple associated with the path can be reduced to the satisfiability of a *trail formula* in Strand*. This formula, when it falls within the syntactic fragment Strand, can be decided using the decision procedure set forth in this paper (see [10] for details on this construction).

It turns out that many annotations for heap-based programs actually fall in the syntactically defined fragment, and so do the verification conditions generated (in fact, all the conditions generated in the experiments in [10] fall into Strand). The annotations fall into Strand as several key properties, such as sortedness, the binary search-tree property, etc., can be stated in this restricted logic with a pure *universal* prefix. Furthermore, the verification condition itself often turns out to be expressible in Strand, as the trail formula introduces variables for the footprint the basic path touches, but these variables are *existentially* quantified, and hence can be related using non-elastic relations (such as the next-node relation). Consequently, Strand is a powerful logic to prove properties of heap manipulating programs.

We now formally define recursively defined data-structures, the notion of *valid subsets* of a tree (which allows us to define submodels), define elastic relations, and define the syntactic fragment Strand.

2.2 Recursively Defined Data-Structures

For any $k \in \mathbb{N}$, let $[k]$ denote the set $\{1, \dots, k\}$. A k-ary tree is a set $V \subseteq [k]^*$, where V is non-empty and prefix-closed. We call $u.i$ the i'th child of u, for every $u, u.i \in V$, where $u \in [k]^*$ and $i \in [k]$. Let us fix a countable set of first-order variables FV (denoted by s, t, etc.), a countable set of set-variables SV (denoted by S, T, etc.), and a countable set of Boolean-variables BV (denoted by p, q, etc.). The syntax of the Monadic second-order (MSO) [20] formulas on k-ary trees is defined:

$$\delta ::= \; p \;\mid\; succ_i(s,t) \;\mid\; s = t \;\mid\; s \in S \;\mid\; \varphi \vee \varphi \;\mid\; \neg\varphi \;\mid\; \exists s.\varphi \;\mid\; \exists S.\varphi \;\mid\; \exists p.\varphi$$

where $i \in [k]$. The atomic formula $succ_i(s,t)$ holds iff t is the i'th child of s. Other logical symbols are interpreted in the traditional way.

Definition 1 (Recursively defined data-structures). *A class of* recursively defined data-structures *over a graph signature $\Sigma = (L_v, L_e)$ (where L_v and L_e are finite sets of labels) is specified by a tuple $\mathcal{R} = (\psi_{Tr}, \psi_U, \{\alpha_a\}_{a \in L_v}, \{\beta_b\}_{b \in L_e})$ where ψ_{Tr} is an MSO sentence, ψ_U is a unary predicate defined in MSO, and each α_a and β_b are monadic and binary predicates defined using MSO, respectively, where all MSO formulas are over k-ary trees, for some $k \in \mathbb{N}$.* □

Let $\mathcal{R} = (\psi_{Tr}, \psi_U, \{\alpha_a\}_{a \in L_v}, \{\beta_b\}_{b \in L_e})$ be a recursively defined data-structure and T be a k-ary Σ-labeled tree that satisfies ψ_{Tr}. Then $T = (V, \{E_i\}_{i \in [k]})$ defines a graph $Graph(T) = (N, E, \mu, \nu, L_v, L_e)$ as follows:

- $N = \{s \in V \mid \psi_U(s)$ *holds in* $T\}$
- $E = \{(s, s') \mid \psi_U(s)$ *and* $\psi_U(s')$ *hold, and* $\beta_b(s, s')$ *holds in* T *for some* $b \in L_e\}$
- $\mu(s) = \{a \in L_v \mid \psi_U(s)$ *holds and* $\alpha_a(s)$ *holds in* $T\}$
- $\nu(s, s') = \{b \in L_e \mid \psi_U(s)$ *and* $\psi_U(s')$ *hold and* $\beta_b(s, s')$ *holds in* $T\}$.

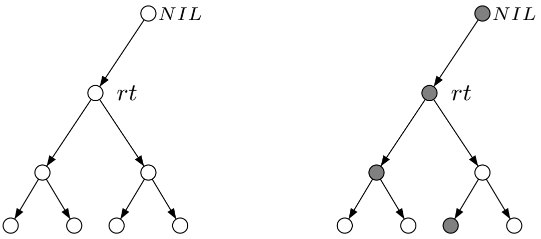
In the above, N denotes the nodes of the graph, E the set of edges, μ the labels on nodes, and ν the labels on edges. The class of graphs defined by $\mathcal{R}$ is the set $Graph(\mathcal{R}) = \{Graph(T) \mid T \models \psi_{Tr}\}$. These graphs are interpreted as heap structures.

We give an examples of modeling heap structures as recursively defined data-structures below.

Example 1. Binary trees are common data-structures. Two field pointers, l and r, point to the left and right children, respectively. If a node does not have a left (right) child, then the l (r) field points to the unique *NIL* node in the heap. Moreover, there is a node rt which is the root of the tree. Binary trees can be modeled as a recursively defined data-structure. For example, we can model the unique *NIL* node as the root of the tree, and model the actual nodes of the binary tree at the left subtree of the root (i.e. the tree under the left child of the root models rt). The right subtree of the root is empty. Binary trees can be modeled as $\mathcal{R}_{bt} = (\psi_{Tr}, \psi_U, \{\alpha_{rt}, \alpha_{nil}\}, \{\beta_l, \beta_r\})$ where

$$\begin{aligned}
\psi_{Tr} &\equiv \exists y_1.\Big(root(y_1) \wedge \nexists y_2.\big(succ_r(y_1, y_2)\big)\Big)\\
\psi_U(x) &\equiv true\\
\alpha_{rt}(x) &\equiv \exists y.\big(root(y) \wedge succ_l(y, x)\big)\\
\alpha_{NIL}(x) &\equiv root(x)\\
\beta_l(x_1, x_2) &\equiv \exists y.\big(root(y) \wedge leftsubtree(y, x_1) \wedge succ_l(x_1, x_2)\big) \vee\\
&\quad \big(root(x_2) \wedge \nexists z.succ_l(x_1, z)\big)\\
\beta_r(x_1, x_2) &\equiv \exists y.\big(root(y) \wedge leftsubtree(y, x_1) \wedge succ_r(x_1, x_2)\big) \vee\\
&\quad \big(root(x_2) \wedge \nexists z.succ_r(x_1, z)\big)
\end{aligned}$$

where the predicate $root(x)$ indicates whether x is the root of the backbone tree, and the relation $leftsubtree(y, x)$ ($rightsubtree(y, x)$) indicates whether x belongs to the subtree of the left (right) child of y. They can all be defined easily in MSO. As an example, Figure 1a shows a binary tree represented in $\mathcal{R}_{bt}$.



(a) T: a binary tree

(b) S: a valid subset of T (shaded nodes)

Fig. 1. A binary tree example represented in $\mathcal{R}_{bt}$

2.3 Submodels

We first define *valid subsets* of a tree, with respect to a recursive data-structure.

Definition 2 (Valid subsets). *Let* $\mathcal{R} = (\psi_{Tr}, \psi_U, \{\alpha_a\}_{a \in L_v}, \{\beta_b\}_{b \in L_e})$ *and* $T = (V, \lambda)$ *be a* Σ*-labeled tree that satisfies* ψ_{Tr}*, and let* $S \subseteq V$*. Then we say* S *is a* valid subset *of* V *if the following hold:*

- *S is non-empty, and least-common-ancestor closed (i.e. for any $s, s' \in S$, the least common ancestor of s and s' wrt T also belongs to S);*
- *The* subtree *defined by S, denoted by Subtree(T, S), is the tree with nodes S, and where the i'th child of a node $u \in S$ is the (unique) node $u' \in S$ closest to u that is in the subtree rooted at the i'th child of u. (This is uniquely defined since S is least-common-ancestor closed.) We require that Subtree(T, S) also satisfy ψ_{Tr};*
- *for every $s \in S$, if $\psi_U(s)$ holds in Subtree(T, S), then $\psi_U(s)$ holds in T as well;*
- *for every $s \in S$, for every $a \in L_v$, $\alpha_a(s)$ holds in Subtree(T, S) iff $\alpha_a(s)$ holds in T.* □

Figure 1b shows a valid subset S of the binary tree representation T in Example 1. A tree $T' = (V', \lambda')$ is said to be a *submodel* of $T = (V, \lambda)$ if there is a valid subset S of V such that T' is isomorphic to *Subtree*(T, S). Note that while unary predicates (α_a) are preserved in the submodel, the edge-relations (β_b) may be very different than its interpretation in the larger model.

Interpreting Formulas on Submodels. We define a transformation $tailor_X$ from an MSO formula on trees to another MSO formula on trees, such that for any MSO sentence δ on k-ary trees, for any tree $T = (V, \lambda)$ and any valid subset $X \subseteq V$, *Subtree*(T, X) satisfies δ iff T satisfies $tailor_X(\delta)$. The transformation is given below, where we let $x \leq y$ mean that y is a descendent of x in the tree. The crucial transformations are the edge-formulas, which are interpreted as the edges of the subtree defined by X.

- $tailor_X(succ_i(s,t)) = \exists s'. \Big(E_i(s, s') \;\wedge\; s' \leq t \;\wedge\; \forall t'. \big((t' \in X \wedge s' \leq t') \Rightarrow t \leq t' \big) \Big)$, for every $i \in [k]$.
- $tailor_X(s = t) = (s = t)$
- $tailor_X(s \in W) = s \in W$
- $tailor_X(\delta_1 \vee \delta_2) = tailor(\delta_1) \vee tailor_X(\delta_2)$
- $tailor_X(\neg \delta) = \neg tailor_X(\delta)$
- $tailor_X(\exists s. \delta) = \exists s. \big(s \in X \wedge tailor_X(\delta) \big)$
- $tailor_X(\exists W. \delta) = \exists W. \big(W \subseteq X \wedge tailor_X(\delta) \big)$

Now by the definition of valid subsets, we define a predicate *ValidSubset*(X) using MSO, where X is a free set variable, such that *ValidSubset*(X) holds in a tree $T = (V, \lambda)$ iff X is a valid subset of V (below, *lca(x,y,z)* stands for an MSO formula says that z is the least common ancestor of x and y in the tree).

$$\begin{aligned} \mathit{ValidSubset}(X) \equiv\ & \forall s,t,u.\Big(\big(s \in X \wedge t \in X \wedge \mathit{lca}(s,t,u)\big) \Rightarrow u \in X\Big) \\ & \wedge\ \mathit{tailor}_X(\psi_{Tr}) \wedge \Big(\forall s.\Big(s \in X \wedge \mathit{tailor}_X\big(\psi_U(s)\big)\Big) \Rightarrow \psi_U(s)\Big) \\ & \wedge \bigwedge_{a \in L_v} \Big[\forall s.\Big(s \in X \Rightarrow \Big(\mathit{tailor}_X\big(\alpha_a(s)\big) \Leftrightarrow \alpha_a(s)\Big)\Big)\Big] \end{aligned}$$

2.4 Elasticity and Strand

Elastic relations are relations of the recursive data-structure that satisfy the property that a pair of nodes satisfy the relation in a tree iff they also satisfy the relation in any valid subtree. Formally,

Definition 3 (Elastic relations). *Let $\mathcal{R} = (\psi_{Tr}, \psi_U, \{\alpha_a\}_{a \in L_v}, \{\beta_b\}_{b \in L_e})$, and let $b \in L_e$ be an edge label. Then the relation E_b (defined by β_b) is* elastic *if for every tree $T = \{V, \lambda\}$ satisfying ψ_{Tr}, for every valid subset S of V, and for every pair of nodes u, v in the model $M' = \mathit{Graph}(\mathit{Subtree}(T, S))$, $E_b(u, v)$ holds in M' iff $E_b(u, v)$ holds in $\mathit{Graph}(T)$.* □

For example, let $\mathcal{R}$ be the class of binary trees, the *left-descendent* relation relating a node with any of the nodes in the tree subtended from the left child, is elastic, because for any binary tree T and any valid subset of S containing nodes x and y, if y is in the left branch of x in T, y is also in the left branch of x in the subtree defined by S, and vise versa. However, the *left-child* relation is non-elastic. Consider a binary tree T in which y is in the left branch of x but not the left child of x, then $S = \{x, y\}$ is a valid subset, and y is the left child of x in $\mathit{Subtree}(T, S)$.

It turns out that elasticity of E_b can also be expressed by the following MSO formula

$$\begin{aligned} \psi_{Tr}\ \Rightarrow\ \forall X\ \forall u\ \forall v.\ \Big[\ & \Big(\mathit{ValidSubset}(X) \wedge u{\in}X \wedge v{\in}X \wedge \\ & \quad \mathit{tailor}_X\big(\psi_U(u)\big) \wedge \mathit{tailor}_X\big(\psi_U(v)\big)\Big) \\ & \Rightarrow\ \Big(\beta_b(u,v) \Leftrightarrow \mathit{tailor}_X\big(\beta_b(u,v)\big)\Big)\ \Big] \end{aligned}$$

E_b is elastic iff the above formula is valid over all trees. Hence, we can *decide* whether a relation is elastic or not, by checking the validity of the above formula over k-ary Σ-labeled trees.

For the rest of this paper, let us fix a class of recursively defined data-structures $\mathcal{R} = (\psi_{Tr}, \psi_U, \{\alpha_a\}_{a \in L_v}, \{\beta_b\}_{b \in L_e})$, with $L_e^E \subseteq L_e$ the set of elastic edge relations, and let $L_e^{NE} = L_e \backslash L_e^E$ the non-elastic edge relations. All notations used are with respect to $\mathcal{R}$.

Formula	$\psi ::= \exists x.\psi \mid \omega$
∀Formula	$\omega ::= \forall y.\omega \mid \varphi$
QFFormula	$\varphi ::= \gamma(e_1, \ldots, e_n) \mid Q_a(v) \mid E_b(v, v') \mid E_{b'}(x, x')$ $\mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2$
Expression	$e ::= data(x) \mid data(y) \mid c \mid g(e_1, \ldots, e_n)$
∃DVar	$x \in \; Loc$
∀DVar	$y \in \; Loc$
DVar	$v ::= x \mid y$
Constant	$c \in \; Sig(\mathcal{D})$
Function	$g \in \; Sig(\mathcal{D})$
$\mathcal{D}$−Relation	$\gamma \in \; Sig(\mathcal{D})$
E − Relation	$b \in \; L_e^E$
NE − Relation	$b' \in \; L_e^{NE}$
Predicate	$a \in \; L_v$

Fig. 2. Syntax of STRAND

STRAND* (called STRAND in [10]) is a two-sorted logic interpreted on program heaps with both locations and the data stored in them. STRAND formulas are of the form $\exists\vec{x}\forall\vec{y}\varphi(\vec{x}, \vec{y})$, where $\vec{x}$ and $\vec{y}$ are ∃DVar and ∀DVar, respectively, (we also refer to both as DVar), φ is a formula that combines structural constraints as well as data-constraints, but their data-constraints are only allowed to refer to $\vec{x}$ and $\vec{y}$. STRAND* is an expressive logic, allowing complex combinations of structural and data constraints. This paper focuses on a decidable fragment STRAND. Given a recursively defined data-structure $\mathcal{R}$ and a first-order theory $\mathcal{D}$, the syntax of STRAND is presented in Figure 2.

Intuitively, STRAND formulas are of the kind $\exists\vec{x}\forall\vec{y}\varphi(\vec{x}, \vec{y})$, where φ is quantifier-free and combines both data-constraints and structural constraints, with the important restriction that *the atomic relations involving universally quantified variables be only elastic relations.*

3 The New Decision Procedure for STRAND

The decision procedure for STRAND presented in [10] worked as follows. Given a STRAND formula ψ over a class of recursively defined data-structures $\mathcal{R}$, we first construct a pure MSO formula on k-ary trees $MinModel_\psi$ that captures the subset of trees that are minimal with respect to an equi-satisfiability preserving embedding relation. This assures that if the formula ψ is satisfiable, then it is satisfiable by a data-extension of a minimal model (a minimal model is a model satisfying $MinModel_\psi$). Furthermore, this set of minimal models was guaranteed to be finite. The decision procedure is then to do a simple analysis on the tree automaton accepting all minimal models, to determine the maximum height h of all minimal trees, and then query the data-solver as to whether any tree of height bounded by h satisfies the STRAND formula.

In this paper, we follow a similar approach, but we replace the notion of minimal models with a new notion called *small models*. Given a STRAND formula $\psi = \exists \vec{x} \forall \vec{y} \varphi(\vec{x}, \vec{y})$ over a class of recursively defined data-structures $\mathcal{R} = (\psi_{Tr}, \psi_U, \{\alpha_a\}_{a \in L_v}, \{\beta_b\}_{b \in L_e})$, the MSO formula *SmallModel*$(\vec{x})$ is defined on k-ary trees, with free variables $\vec{x}$. Intuitively, *SmallModel*$(\vec{x})$ says that there does not exist a nontrivial valid subset X such that X contains $\vec{x}$, and further satisfies the following: for every non-elastic relation possibly appearing in $\varphi(\vec{x}, \vec{y})$, it holds in *Graph*$(T)$ iff it holds in *Graph*$($*Subtree*$(T, X))$. Since the evaluations of other atomic formulas, including elastic relations and data-logic relations, are all preserved, we can prove that *SmallModel*$(\vec{x})$ is equisatisfiable to the structural constraints in ψ, but has only a finite number of models. The formula *SmallModel*$(\vec{x})$ is defined as follows:

$$\begin{aligned}
\mathit{SmallModel}(\vec{x}) \quad \equiv \quad & \psi_{Tr} \ \wedge \ \bigwedge_{x \in \vec{x}} \psi_U(x) & (\zeta) \\
\wedge \quad & \neg \exists X. \Big(\mathit{ValidSubset}(X) \ \wedge \ \bigwedge_{x \in \vec{x}} (x \in X) \ \wedge \\
& \qquad \exists s. (s \in X) \ \wedge \ \exists s. (s \notin X) \ \wedge & (\eta) \\
& \qquad \bigwedge_{b \in L_e^{NE}, x, x' \in \vec{x}} \Big(\beta_b(x, x') \Leftrightarrow \mathit{tailor}_X \big(\beta_b(x, x') \big) \Big) \Big)
\end{aligned}$$

Note that the above formula *does not depend* on the STRAND formula ψ at all, except for the set of existentially quantified variables $\vec{x}$.

Our proof strategy is now as follows. We show two technical results:

(a) For any $\vec{x}$, *SmallModel*$(\vec{x})$ has only finitely many models (Theorem 1 below). This result is independent of the fact that we are dealing with STRAND formulas.

(b) A STRAND formula ψ with existentially quantified variables $\vec{x}$ has a model iff there is some data-extension of a model satisfying *SmallModel*$(\vec{x})$ that satisfies ψ (Theorem 2 below).

The above two establish the correctness of the decision procedure. Given a STRAND formula ψ, with existential quantification over $\vec{x}$, we can compute a tree-automaton accepting the set of all small models (i.e. the models of *SmallModel*$(\vec{x})$), compute the maximum height h of the small models, and then query the data-solver as to whether there is a model of height at most h *with data* that satisfies ψ.

We now prove the two technical results.

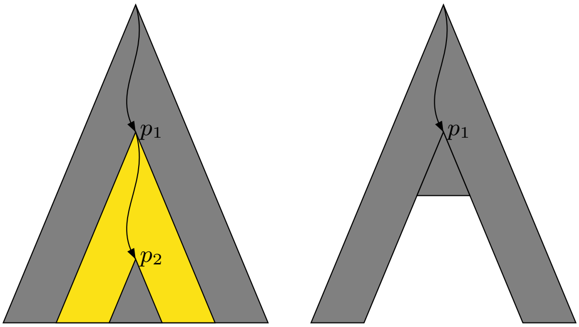
Theorem 1. *For any recursively defined data-structure $\mathcal{R}$ and any finite set of variables $\vec{x}$, the number of models of SmallModel$(\vec{x})$ is finite.*

Proof. Fix a recursively defined data-structure $\mathcal{R}$ and a finite set of variables $\vec{x}$. It is sufficient to show that for any model T of $\mathit{SmallModel}(\vec{x})$, the size of T is bounded.

We first split $\mathit{SmallModel}(\vec{x})$ into two parts: let ζ be the first two conjuncts, i.e., $\psi_{Tr} \wedge \bigwedge_{x \in \vec{x}} \psi_U(x)$, and η be the last conjunct.

Recall the classic logic-automata connection: for any MSO formula $\theta(\vec{y}, \vec{Y})$ with free first-order variables $\vec{y}$ and free set-variables $\vec{Y}$, we can construct a tree-automaton that precisely accepts those trees with encodings of the valuation of $\vec{y}$ and $\vec{Y}$ as extra labels that satisfy the formula θ [20].

Construct a deterministic (bottom-up) tree automaton A_ζ that accepts precisely the models satisfying $\zeta(\vec{x})$, using this classic logic-automata connection [20]. Also, for each *non-elastic* edge label $b \in L_r^{NE}$, and each pair of variables $x, x' \in \vec{x}$, let $A_{b,x,x'}$ be a deterministic (bottom-up) tree automaton that accepts the models of the formula $\beta_b(x, x')$.



(a) T with the valid subset X (shaded dark) **(b)** $\mathit{Subtree}(T, X)$

Fig. 3. A valid subset X that falsifies β

It is clear that T is accepted by A_ζ, while $A_{b,x,x'}$, for each b, x, x', either accepts or rejects T. Construct the *product* of the automaton A_ζ and all automata $A_{b,x,x'}$, for each b, x, x', with the acceptance condition derived solely from A_ζ; call this automaton B; then B accepts T.

If the accepted run of B on T is r, then we claim that r is *loop-free* (a run of a tree automaton is loop-free if for any path of the tree, there are no two nodes in the path labeled by the same state). Assume not. Then there must be two different nodes p_1, p_2 such that p_2 is in the subtree of p_1, and both p_1 and p_2 are labeled by the same state q in r. Then we can *pump down* T by merging p_1 and p_2. The resulting tree is accepted by A_T as well. Consider the subset X of T that consists of those remaining nodes, as shown in Figure 3. It is not hard to see X is a nontrivial valid subset of T. Also for each $b \in L_r^{NE}$ and each $x, x' \in \vec{x}$, since the run of $A_{b,x,x'}$ ends up in the same state on reading the subtree corresponding to X, $\beta_b(x, x')$ holds in T iff $\beta_b(x, x')$ holds in $\mathit{Subtree}(T, X)$. Thus

X is a valid subset of T that acts to falsify η, which contradicts our assumption that T satisfies $\zeta \wedge \eta$.

Since r is loop-free, the height of T is bounded by the number of states in B. □

We now show that the small models define an adequate set of models to check for satisfiability.

Theorem 2. *Let $\mathcal{R}$ be a recursively defined data-structure and let $\psi = \exists\vec{x}\forall\vec{y}\ \varphi(\vec{x},\vec{y})$ be a* STRAND *formula. If ψ is satisfiable, then there is a model $\mathcal{M}$ of ψ and a model T of SmallModel($\vec{x}$) such that Graph(T) is isomorphic to the graph structure of $\mathcal{M}$.*

Proof. Let ψ be satisfiable and let $\mathcal{M}$ satisfy ψ. Then there is an assignment I of $\vec{x}$ over the nodes of $\mathcal{M}$ under which $\forall\vec{y}\varphi(\vec{x},\vec{y})$ is satisfied.

Let T be the backbone tree of the graph model of $\mathcal{M}$, and further let us add an extra label over T to denote the assignment I to $\vec{x}$.

Let us, without loss of generality, assume that T is a *minimal* tree; i.e. T has the least number of nodes among all models satisfying ψ.

We claim that T satisfies *SmallModel*($\vec{x}$) under the interpretation I.

Assume not, i.e., T does not satisfy *SmallModel*($\vec{x}$) under I. Since T under I satisfies ζ, it must not satisfy η. Hence there exists a strict valid subset of nodes, X, such that every non-elastic relation holds over every pair of variables in $\vec{x}$ the same way in T as it does on the subtree defined by X.

Let $\mathcal{M}'$ be the model obtained by taking the graph of the subtree defined by X as the underlying graph, with data at each obtained node inherited from the corresponding node in $\mathcal{M}$. We claim that $\mathcal{M}'$ satisfies ψ as well, and since the tree corresponding to $\mathcal{M}'$ is a strict subtree of T, this contradicts our assumption on T.

We now show that the graph of the subtree defined by X has a data-extension that satisfies ψ.

In order to satisfy ψ, we take the interpretation of each $x \in \vec{x}$ to be the node in $\mathcal{M}'$ corresponding to $I(x)$. Now consider any valuation of $\vec{y}$. We will show that every *atomic* relation in φ holds in $\mathcal{M}$ in precisely *the same way* as it does on $\mathcal{M}'$; this will show that φ holds in $\mathcal{M}$ iff φ holds in $\mathcal{M}'$, and hence that φ holds in $\mathcal{M}'$.

By definition, an atomic relation τ could be a unary predicate, an elastic binary relation, a non-elastic binary relation, or an atomic data-relation. If τ is a unary predicate $Q_a(v)$ (where $v \in \vec{x} \cup \vec{y}$), then by definition of submodels (and valid subsets), τ holds in $\mathcal{M}'$ iff τ holds in $\mathcal{M}$. If τ is an elastic relation $E_b(v_1, v_2)$, by definition of elasticity, τ holds in $\mathcal{M}$ iff $\beta_b(v_1, v_2)$ holds in T iff $\beta_b(v_1, v_2)$ holds in *Subtree*(T, X) iff $\tau(v_1, v_2)$ holds in $\mathcal{M}'$. If τ is a non-elastic relation, it must be of form $E_b(x, x')$ where $x, x' \in \vec{x}$. By the properties of X established above, it follows that $E_b(x, x')$ holds in $\mathcal{M}'$ iff $E_b(x, x')$ holds in $\mathcal{M}$. Finally, if τ is an atomic data-relation, since the data-extension of $\mathcal{M}'$ is inherited from $\mathcal{M}$, the data-relation holds in $\mathcal{M}'$ iff it holds in $\mathcal{M}$.

The contradiction shows that $\mathcal{M}$ is a small model. □

The above results can be used to even *pre-compute* the bounds on sizes of the structural models for a fixed recursive data-structure and for various numbers of existentially quantified variables, and *completely* avoid the structural phase altogether. We can even establish these bounds *analytically* (and manually), without the use of a solver, for some data-structures. For instance, over trees, with non-elastic relations *left-child* and *right-child*, and other elastic relations, it is not hard to see that a STRAND formula is satisfiable iff it is satisfiable by a tree of size at most $2n$, where n is the number of existentially quantified variables in the formula.

3.1 Comparison with Earlier Known Decision Procedure

We now compare the new decision procedure, technically, with the earlier known decision procedure for STRAND [10], which was also the decision procedure for the semantic decidable fragment $\text{STRAND}_{dec}^{sem}$.

The known decision procedure for STRAND [10] worked as follows. Given a STRAND formula, we first eliminate the leading existential quantification, by absorbing it into the signature, using new constants. Then, for the formula $\forall\vec{y}\varphi$, we define a *structural abstraction* of φ, named $\widehat{\varphi}$, where all data-predicates are replaced uniformly by a set of Boolean variables $\vec{p}$. A model that satisfies $\widehat{\varphi}$, for every valuation of $\vec{y}$, using some valuation of $\vec{p}$ is said to be a *minimal model* if it has no proper submodel that satisfies $\widehat{\varphi}$ under the *same* valuation $\vec{p}$, for every valuation of $\vec{y}$. Intuitively, this ensures that if the model can be populated with data in some way so that $\forall\vec{y}\varphi$ is satisfied, then the submodel satisfy the formula $\forall\vec{y}\varphi$ as well, by inheriting the same data-values from the model.

It turns out that for any STRAND formula, the number of minimal models (with respect to the submodel relation) is *finite* [10]. Moreover, we can capture the class of all minimal models using an MSO formula of the following form:

$$\begin{aligned} \mathit{MinModel} = \neg\exists X.\Big[\ & \mathit{ValidSubset}(X)\ \wedge\ \exists s.(s \in X) \wedge \exists s.(s \notin X) \wedge \\ & \forall\vec{y}\ \forall\vec{p}\ \Big(\Big(\ \wedge_{y\in\vec{y}}\ \big(y \in X \wedge \psi_U(y)\big) \wedge \mathit{interpret}(\widehat{\varphi}(\vec{y},\vec{p}))\Big) \\ & \qquad \Rightarrow \mathit{tailor}_X\big(\mathit{interpret}(\widehat{\varphi}(\vec{y},\vec{p}))\big)\Big)\ \Big] \end{aligned}$$

The above formula intuitively says that a model is minimal if there is no valid non-trivial submodel such that for all possible valuations of $\vec{y}$ in the submodel, and all possible valuations of $\vec{p}$, the model satisfies the structural abstraction $\widehat{\varphi}(\vec{y},\vec{p})$, then so does the submodel.

We can enumerate all the minimal models (all models satisfying the above formula), and using a data-constraint solver, ask whether any of them can be extended with data to satisfy the STRAND formula. Most importantly, if none of them can, we know that the formula is unsatisfiable (for if there was a model that satisfies the formula, then one of the minimal models will be a submodel that can inherit the data values and satisfy the formula).

Comparing the above formula to the formula for *SmallModel*, notice that the latter is incredibly simpler as it does not refer to the STRAND formula (i.e. φ) at all! The *SmallModel* formula just depends on the *set of existentially quantified variables* and the non-elastic relations in the signature. In contrast, the formula above for *MinModel* uses adaptations of the STRAND formula φ *twice* within it. In practice, this results in a very complex formula, as the verification conditions get long and involved, and this results in poor performance by the MSO solver. In contrast, as we show in the next section, the new procedure results in simpler formulas that get evaluated significantly faster in practice.

4 Experiments

In this section, we demonstrate the efficiency of the new decision procedure for STRAND by checking verification conditions of several heap-manipulating programs, and comparing them to the decision procedure in [10]. These examples include list-manipulating and tree-manipulating programs, including searching a sorted list, inserting into a sorted list, in-place reversal of a sorted list, the bubble-sort algorithm, searching for a key in a binary search tree, inserting into a binary search tree, and doing a left- or right-rotate on a binary search tree.

For all these examples, a set of partial correctness properties including both structural and data requirements is checked. For example, assuming a node with value k exists, we check if both `sorted-list-search` and `bst-search` return a node with value k. For `sorted-list-insert`, we assume that the inserted value does not exist, and check if the resulting list contains the inserted node, and the sortedness property continues to hold. In the program `bst-insert`, assuming the tree does not contain the inserted node in the beginning, we check whether the final tree contains the inserted node, and the binary-search-tree property continues to hold. In `sorted-list-reverse`, we check if the output list is a valid list that is reverse-sorted. The code for `bubblesort` is checked to see if it results in a sorted list. And the `left-rotate` and `right-rotate` codes are checked to see whether they maintain the binary search-tree property.

In the structural solving phase using MONA, when a STRAND formula ψ is given, we further optimize the formula $\mathit{SmallModel}(\vec{x})$ with respect to ψ for better performance, as follows. First, a sub-formula $\beta_b(x, x') \Leftrightarrow \mathit{tailor}_X(\beta_b(x, x'))$ appears in the formula only if the atomic formula $E_b(x, x')$ appears in ψ. Moreover, if $E_b(x, x')$ only appears positively, we use $\beta_b(x, x') \Rightarrow \mathit{tailor}_X(\beta_b(x, x'))$ instead; similarly if $E_b(x, x')$ occurs only negatively, then we use $\beta_b(x, x') \Leftarrow \mathit{tailor}_X(\beta_b(x, x'))$ instead. This is clearly sound.

Figure 4 shows the comparison of the two decision procedures on checking the verification conditions. The results for both procedures were conducted on the same 2.2GHz, 4GB machine running Windows 7. We also report the size of the largest intermediate BDD and the time spent by MONA.

The experimental results show a magnitude of speed-up, with some examples (like `sorted-list-insert/after-loop`) giving even a 1000X speedup. The peak BDD sizes are also considerably smaller, in general, using the new algorithm.

Program	Verification condition	Minimal Model computation (old Alg. [10])		Small Model computation (new Alg.)		Data-constraint solving (Z3, QF-LIA)
		Max. BDD size	Time(s)	Max. BDD size	Time(s)	Old/New Time (s)
sorted-list-search	before-loop	10009	0.34	**540**	**0.01**	-
	in-loop	17803	0.59	**12291**	**0.14**	-
	after-loop	3787	0.18	**540**	**0.01**	-
sorted-list-insert	before-head	59020	1.66	**242**	**0.01**	0.02/0.02
	before-loop	15286	0.38	**595**	**0.01**	-
	in-loop	135904	4.46	**3003**	**0.03**	-
	after-loop	475972	13.93	**1250**	**0.01**	0.02/0.03
sorted-list-insert-error	before-loop	14464	0.34	**595**	**0.01**	0.02/0.02
sorted-list-reverse	before-loop	2717	0.24	**1155**	**0.01**	-
	in-loop	89342	2.79	**12291**	**0.14**	-
	after-loop	3135	0.35	**1155**	**0.01**	-
bubblesort	loop-if-if	179488	7.70	**73771**	**1.31**	-
	loop-if-else	155480	6.83	**34317**	**0.48**	-
	loop-else	95181	2.73	**7017**	**0.07**	0.02/0.04
bst-search	before-loop	9023	5.03	**1262**	**0.31**	-
	in-loop	26163	32.80	**3594**	**2.43**	0.02/0.11
	after-loop	6066	3.27	**1262**	**0.34**	-
bst-insert	before-loop	3485	1.34	**1262**	**0.34**	-
	in-loop	17234	9.84	**1908**	**1.38**	-
	after-loop	2336	1.76	**1807**	**0.46**	-
left-/right-rotate	bst-preserving	**1086**	1.59	1510	**0.48**	0.05/0.14
Total			98.15		**7.99**	0.15/0.36

Fig. 4. Results of program verification (details at `www.cs.uiuc.edu/∼qiu2/strand`)

Turning to the *bounds* computed on the minimal/small models, the two procedures generate the same bounds (i.e. the same lengths for lists and the same heights for trees) on all examples, except `bst-search-in-loop` and `left-rotate`. The condition `bst-search-in-loop` gave a maximal height of 5 with the old procedure and maximal height 6 in the new, while `left-rotate` generates maximal height 5 in the old procedure and 6 in the new. However, the larger bounds did not affect the data solving phase by any significant degree (at most by 0.1s)

The experiments show that the new algorithm is considerably more efficient than the earlier known algorithm on this set of examples.

5 Related Work

Apart from [10], the work closest to ours is Pale [13], which is a logic on heaps structures but not data, and uses MSO and Mona [6] to decide properties of heaps. Tasc [5] is similar but generalizes to reason balancedness in the cases of

AVL and red-black trees. First-order logics with axiomatizations of the reachability relation (which cannot be expressed in FOL) have been proposed: axioms capturing *local* properties [12], a logic on regular patterns that is decidable [21], among others.

The logic in Havoc, called Lisbq [9], offers reasoning with generic heaps combined with an arbitrary data-logic. The logic has restricted reachability predicates and universal quantification, but is syntactically severely curtailed to obtain decidability. Though the logic is not very expressive, it is extremely efficient, as it uses no structural solver, but translates the structure-solving also to the (Boolean aspect) of the SMT solver. The logic Csl [4] is defined in a similar vein as Havoc, with similar sort-restrictions on the syntax, but generalizes to handle doubly-linked lists, and allows size constraints on structures. As far as we know, neither Havoc nor Csl can express the verification conditions of searching a binary search tree. The work reported in [3] gives a logic that extends an LTL-like syntax to define certain decidable logic fragments on heaps.

The inference rule system proposed in [16] for reasoning with restricted reachability does not support universal quantification and cannot express disjointness constraints, but has an SMT solver based implementation [17]. Restricted forms of reachability were first axiomatized in early work by Nelson [14]. Several mechanisms without quantification exist, including the work reported in [18,1]. Kuncak's thesis describes automatic decision procedures that approximate higher-order logic using first-order logic, through approximate logics over sets and their cardinalities [8].

Finally, separation logic [19] is a convenient logic to express heap properties of programs, and a decidable fragment (without data) on lists is known [2]. However, not many extensions of separation logics support data constraints (see [11] for one that does).

6 Conclusions

The decision procedures for Strand use a structural phase that computes a set of minimal structural models in a completely data-logic agnostic manner [10]. The new decision procedure set forth in this paper gives a way of computing an equisatisfiable finite set of structural models that is even agnostic to the Strand formula. This yields a much simpler decision procedure, in theory, and a much faster decision procedure, in practice.

We are emboldened by the very minimal reliance on the structural solver (Mona) and we believe that the approach described in this paper is ready to be used for generating unit test inputs for heap-manipulating methods using symbolic constraint solving. Implementing such a procedure, as well as implementing a fully-fledged solver for Strand, are interesting future directions to pursue.

Acknowledgements. This work is partially funded by NSF CAREER award #0747041.

References

1. Balaban, I., Pnueli, A., Zuck, L.D.: Shape analysis of single-parent heaps. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 91–105. Springer, Heidelberg (2007)
2. Berdine, J., Calcagno, C., O'Hearn, P.W.: A decidable fragment of separation logic. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2004. LNCS, vol. 3328, pp. 97–109. Springer, Heidelberg (2004)
3. Bjørner, N., Hendrix, J.: Linear functional fixed-points. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 124–139. Springer, Heidelberg (2009)
4. Bouajjani, A., Drăgoi, C., Enea, C., Sighireanu, M.: A logic-based framework for reasoning about composite data structures. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 178–195. Springer, Heidelberg (2009)
5. Habermehl, P., Iosif, R., Vojnar, T.: Automata-based verification of programs with tree updates. Acta Informatica 47(1), 1–31 (2010)
6. Klarlund, N., Møller, A.: MONA. BRICS, Department of Computer Science, Aarhus University (January 2001), http://www.brics.dk/mona/
7. Klarlund, N., Schwartzbach, M.I.: Graph types. In: POPL 1993, pp. 196–205. ACM, New York (1993)
8. Kuncak, V.: Modular Data Structure Verification. Ph.D. thesis, Massachusetts Institute of Technology (2007)
9. Lahiri, S., Qadeer, S.: Back to the future: revisiting precise program verification using SMT solvers. In: POPL 2008, pp. 171–182. ACM, New York (2008)
10. Madhusudan, P., Parlato, G., Qiu, X.: Decidable logics combining heap structures and data. In: POPL 2011, pp. 611–622. ACM, New York (2011)
11. Magill, S., Tsai, M.H., Lee, P., Tsay, Y.K.: THOR: A tool for reasoning about shape and arithmetic. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 428–432. Springer, Heidelberg (2008)
12. McPeak, S., Necula, G.C.: Data structure specifications via local equality axioms. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 476–490. Springer, Heidelberg (2005)
13. Møller, A., Schwartzbach, M.I.: The pointer assertion logic engine. In: PLDI 2001, pp. 221–231. ACM, New York (2001)
14. Nelson, G.: Verifying reachability invariants of linked structures. In: POPL 1983, pp. 38–47. ACM, New York (1983)
15. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Trans. Program. Lang. Syst. 1, 245–257 (1979)
16. Rakamarić, Z., Bingham, J.D., Hu, A.J.: An inference-rule-based decision procedure for verification of heap-manipulating programs with mutable data and cyclic data structures. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 106–121. Springer, Heidelberg (2007)
17. Rakamarić, Z., Bruttomesso, R., Hu, A.J., Cimatti, A.: Verifying heap-manipulating programs in an SMT framework. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 237–252. Springer, Heidelberg (2007)
18. Ranise, S., Zarba, C.: A theory of singly-linked lists and its extensible decision procedure. In: SEFM 2006, pp. 206–215. IEEE-CS, Los Alamitos (2006)
19. Reynolds, J.: Separation logic: a logic for shared mutable data structures. In: LICS 2002, pp. 55–74. IEEE-CS, Los Alamitos (2002)
20. Thomas, W.: Languages, automata, and logic. In: Handbook of Formal Languages, pp. 389–456. Springer, Heidelberg (1997)
21. Yorsh, G., Rabinovich, A.M., Sagiv, M., Meyer, A., Bouajjani, A.: A logic of reachable patterns in linked data-structures. In: Aceto, L., Ingólfsdóttir, A. (eds.) FoSSaCS 2006. LNCS, vol. 3921, pp. 94–110. Springer, Heidelberg (2006)

The Flow-Insensitive Precision of Andersen's Analysis in Practice

Sam Blackshear[1], Bor-Yuh Evan Chang[1],
Sriram Sankaranarayanan[1], and Manu Sridharan[2]

[1] University of Colorado Boulder
{samuel.blackshear,evan.chang,sriram.sankaranarayanan}@colorado.edu
[2] IBM T.J. Watson Research Center
msridhar@us.ibm.com

Abstract. We present techniques for determining the precision gap between Andersen's points-to analysis and precise flow-insensitive points-to analysis in practice. While previous work has shown that such a gap may exist, no efficient algorithm for precise flow-insensitive analysis is known, making measurement of the gap on real-world programs difficult. We give an algorithm for precise flow-insensitive analysis of programs with finite memory, based on a novel technique for refining any points-to analysis with a search for flow-insensitive witnesses. We give a compact symbolic encoding of the technique that enables computing the search using a tuned SAT solver. We also present extensions of the algorithm that enable computing lower and upper bounds on the precision gap in the presence of dynamic memory allocation. In our experimental evaluation over a suite of small- to medium-sized C programs, we never observed a precision gap between Andersen's analysis and the precise analysis. In other words, Andersen's analysis computed a precise flow-insensitive result for all of our benchmarks. Hence, we conclude that while better algorithms for the precise flow-insensitive analysis are still of theoretical interest, their practical impact for C programs is likely to be negligible.

1 Introduction

Programming languages such as C and Java make extensive use of pointers. As a result, many program analysis questions over these languages require pointer analysis as a primitive to find the set of all memory locations that a given pointer may address. This problem is of fundamental importance and has been widely studied using numerous approaches [8]. Recently, Andersen's analysis [1] has been increasingly employed to analyze large programs [7, 19]. However, it is also well known that Andersen's analysis falls short of being a *precise* flow-insensitive analysis [5, 9, 17]. A precise flow-insensitive analysis reports only the points-to relationships that are realizable via executing some sequence of program statements, assuming arbitrary control flow between statements. There are two key reasons for the precision gap between Andersen's analysis and a precise flow-insensitive analysis (discussed further in Sect. 2):

E. Yahav (Ed.): SAS 2011, LNCS 6887, pp. 60–76, 2011.

- Andersen's analysis assumes that any set of points-to edges can occur *simultaneously*, whereas program variables must point to a single location at any program state. This discrepancy may cause Andersen's to generate spurious points-to edges.
- Andersen's analysis transforms pointer assignments to contain at most one dereference by rewriting complex statements using fresh temporary variables. However, temporary variables can introduce spurious points-to edges [9].

These observations lead to two tantalizing and long-standing questions:

1. Is there an *efficient* algorithm for precise flow-insensitive pointer analysis?
2. Does a precision gap exist, *in practice*, for real-world programs?

Regarding the first question, precise flow-insensitive analysis is NP-hard for arbitrary finite-memory programs [9], and no polynomial-time algorithm is known even for programs with only Andersen-style statements [5]. In the presence of dynamic memory, the decidability of the problem remains unknown [5].

This paper addresses the second question by presenting techniques for computing the precision gap between Andersen's and precise flow-insensitive points-to analysis in practice. We introduce an algorithm for computing the precise flow-insensitive analysis for programs with finite memory. This algorithm refines Andersen's analysis results by searching for an appropriate sequence of statements to *witness* each edge in the points-to graph obtained from Andersen's analysis. The search is encoded symbolically and carried out using efficient modern SAT solvers. Although the worst-case performance of our algorithm is exponential, our SAT encoding enables analysis of medium-sized C programs within reasonable time/memory bounds. We then extend our techniques to investigate the precision gap in the presence of dynamic memory.

We performed an experimental evaluation to measure the precision gap between Andersen's and precise flow-insensitive analysis on a suite of C programs. Perhaps surprisingly, we found the results of the two analyses to be identical over our benchmarks: *a precision gap seems to be non-existent, in practice.* Thus, we conclude that better algorithms for precise flow-insensitive points-to analysis, while retaining theoretical interest, are unlikely to have a large impact on the analysis of C programs. Instead, our conclusions suggest efforts spent on refining Andersen's analysis with flow or context sensitivity may be more fruitful. Interestingly, our witness search algorithm may offer a basis for such efforts.

This paper makes the following contributions:

- We present an algorithm for precise flow-insensitive analysis for programs with finite memory based on refining Andersen's analysis with a *witness search* for each computed points-to fact (Sect. 3.1).
- We describe extensions for handling dynamic memory over- and under-approximately in order to evaluate the precision gap resulting from the lack of a fully precise treatment of dynamic memory (Sect. 3.2).
- We also give a compact symbolic encoding of the witness search algorithm, enabling the use of highly-tuned SAT solvers for the search (Sect. 3.3).

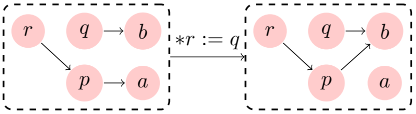
- We implemented our algorithms and performed an experimental evaluation, showing that the precision gap seems non-existent for small- to medium-sized C programs (Sect. 4).

2 Flow-Insensitive Imprecision in Andersen's Analysis

In this section, we examine the sources of imprecision in Andersen's analysis compared to a precise flow-insensitive points-to analysis. Most of this discussion is a reformulation of known concepts.

We first define the notion of a precise flow-insensitive points-to analysis. A (flow-insensitive) points-to analysis problem consists of a finite set of variables X along with a set of assignments A. The simplest variant considers only *finite memory*, that is, each assignment has one of the following forms: $*^d p := \&q$ or $*^{d_1} p := *^{d_2} q$ where p and q are variables. The expression $*^d p$ denotes the application of $d \geq 0$ dereferences to pointer p, while $\&q$ takes the address of q. Note that $*^0 p$ is the same as p. The *dynamic memory* points-to analysis problem adds a statement $*^d p :=$ `malloc`() for allocation. The goal of a precise flow-insensitive points-to analysis is to answer queries of the form $p \mapsto q$: is there a sequence of assignments from A that causes p to point to q (i.e., that causes variable p to contain the address of q)? The problem is flow-insensitive, as program control flow is ignored to produce a set of assignments as input.

The result of a points-to analysis can be captured as a *points-to graph*. A points-to graph $G\colon (V, E)$ consists of a set of vertices V and directed edges E. The set of vertices represents memory cells and thus includes the program variables (i.e., $V \supseteq X$). To conservatively model constructs like aggregates (e.g., arrays or structures), dynamically allocated memory, and local variables in a recursive context, a vertex may model more than one concrete memory cell (which is referred to as a *summary location*). An edge $v_1 \mapsto v_2$ says that v_1 may point to v_2 (i.e., a concrete cell represented by v_1 may contain an address from v_2) under some execution of assignments drawn from A. For convenience, we use the notation $V(G)$ or $E(G)$ to indicate the vertices and edges of G.



An exact abstraction of a concrete memory configuration can be modeled by a points-to graph where each vertex represents a single memory cell and thus each vertex can have at most one outgoing points-to edge. We call such graphs *exact points-to graphs.* A points-to graph obtained as a result of some may points-to analysis may be viewed as the join of some number of exact points-to graphs. With exact point-to graphs, we can define the operational semantics of pointer assignments from a points-to analysis problem. We write $G \xrightarrow{a} G'$ for the one-step transition relation that says assignment a transforms exact graph G to exact graph G'. A formal definition is provided in our companion technical report [3]. The inset figure illustrates the transformation of an exact points-to graph through an assignment. We can now define *realizability* of points-to edges.

Definition 1 (Realizable Graphs, Edges, and Subgraphs). *A graph G is* realizable *iff there exists a sequence of assignments $a_1, \ldots, a_N$ such that $G_0 \xrightarrow{a_1} G_1 \to \cdots \xrightarrow{a_N} G_N \equiv G$ where $G_0\colon (X, \emptyset)$ is the* initial graph *of the points-to-analysis problem with variables X. An edge $v_1 \mapsto v_2 \in V \times V$ is* realizable *iff there exists a realizable graph G such that $v_1 \mapsto v_2 \in E(G)$. A subset of edges $E \subseteq V \times V$ is (simultaneously) realizable if there exists a realizable graph G such that $E \subseteq E(G)$.*

A *precise flow-insensitive points-to analysis* derives all edges that are realizable and no other edges.

Andersen's analysis [1], well studied in the literature, is an over-approximate flow-insensitive points-to analysis computable in polynomial time. In essence, Andersen's analysis works by deriving a graph with all points-to relations using the inference rules shown below:

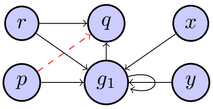
$$\frac{p := \&q}{p \mapsto q} \qquad \frac{p := q \quad q \mapsto r}{p \mapsto r} \qquad \frac{p := *q \quad q \mapsto r \quad r \mapsto s}{p \mapsto s} \qquad \frac{*p := q \quad p \mapsto r \quad q \mapsto s}{r \mapsto s}$$

where an assignment a (e.g., $p := \&q$) in the rule states that a is in the set of program assignments A and a points-to edge e (e.g., $p \mapsto q$) states that e is in the derived points-to graph G (i.e., $e \in E(G)$). Observe that Andersen's analysis requires that an input problem be transformed so that all statements contain at most one dereference. This transformation itself may introduce imprecision, as we shall discuss shortly. Finally, Andersen's analysis handles dynamic memory over-approximately by essentially translating each statement $p :=$ `malloc()` into $p := \&m_i$, where m_i is a fresh *summary location* representing all memory allocated by the statement.

Imprecision: Simultaneous Points-To. Previous work has pointed out that Andersen's is not a precise flow-insensitive points-to analysis [5, 9]. One source of imprecision in Andersen's analysis is a lack of reasoning about what points-to relationships can hold *simultaneously* in possible statement sequences.

Example 1. Consider the following set of pointer assignments:

$$\{p := *r,\ r := \&q,\ r := *x,\ x := \&g_1,\ y := x,\ *x := r,\ *x := y\}\,.$$

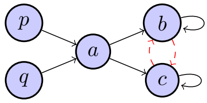


The inset figure shows the Andersen's analysis result for this example (for clarity, graphs with outlined blue nodes are used for analysis results). Notice that while $r \mapsto g_1$ and $g_1 \mapsto q$ are individually realizable, they cannot be realized simultaneously in any statement sequence, as this would require either: (1) pointer r to point to g_1 and q simultaneously; or (2) pointer g_1 to point to g_1 and q simultaneously (further illustration in Sect. 3.1). Andersen's does not consider simultaneous realizability, so with given the statement $p := *r$ and the aforementioned points-to edges, the analysis concludes that p may point to q (shown dashed in red), when in fact this edge is not realizable. The finite heap abstraction employed by Andersen's analysis may lead to conflation of multiple heap pointers, possibly worsening the simultaneous realizability issue.

Imprecision: Program Transformation. Imprecision may also be introduced due to the requisite decomposition of statements with multiple dereferences:

Example 2. Consider the following set of pointer assignments: $\{a := \&b,\ a := \&c,\ p := \&a,\ q := \&a,\ **p := *q\}$. The statement $**p := *q$ may make either b or c point to itself, but in no statement sequence can it make b point to c (as shown in the inset below). However, when decomposed for Andersen's analysis, the statement is transformed into statements introducing fresh variables t_1 and t_2: $t_1 := *p,\ t_2 := *q,\ *t_1 := t_2$. Then, the following sequence causes $b \mapsto c$:

$$a := \&b;\ p := \&a;\ t_1 := *p;\ a := \&c;\ q := \&a;\ t_2 := *q;\ *t_1 := t_2;$$



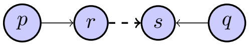
Hence, the transformation to Andersen's-style statements may create additional realizable points-to relationships among the original variables (i.e., the transformation adds imprecision even for precise flow-insensitive analysis). The goal of this work is to determine whether simultaneous realizability or program transformation issues cause a precision gap, in practice.

3 Precise Analysis via Witness Search

In this section, we present a witness search algorithm that yields a precise flow-insensitive points-to analysis for the finite-memory problem (Sect. 3.1). Then, we discuss two extensions to the algorithm that respectively provide over- and under-approximate handling of dynamic memory allocation and other summarized locations (Sect. 3.2). Finally, we describe a SAT-encoding of the search algorithm that yields a reasonably efficient implementation in practice (Sect. 3.3).

3.1 A Precise Algorithm for Finite Memory

Here, we describe our witness search algorithm, which computes a precise flow-insensitive analysis for programs with finite memory. Given the result of a conservative flow-insensitive points-to analysis, such as Andersen's [1], we first create *edge dependency rules* that capture ways a points-to edge may arise. These edge dependency rules are effectively instantiations of the Andersen inference rules. Next, we search for witness sequences for a given edge, on demand, using the edge dependency rules while taking into account constraints on simultaneous realizability. We may find no witness for an edge, in which case we have a *refutation* for the realizability of that points-to fact. Essentially, the dependency rules leverage the Andersen result to constrain a goal-directed search for realizability.



Generating Edge Dependency Rules. We first illustrate edge dependency rule construction through an example. Let G be a points-to graph derived as the result of a conservative points-to analysis. Consider the assignment a: $*p := q$, wherein edges $p \mapsto r$, $q \mapsto s$, and $r \mapsto s$ exist in G (as illustrated inset). In terms of realizability, the following claim can be made in this situation:

Edge $r \mapsto s$ is realizable (using assignment a) if the edge set $\{p \mapsto r,\ q \mapsto s\}$ is simultaneously realizable.

Note that the converse of this statement need not be true—the edge $r \mapsto s$ may be realizable using another set of edges and/or a different pointer assignment. In our framework, this assertion is represented by a *dependency rule*:

$$r \mapsto s \xleftarrow{a:\ *p:=q} \{p \mapsto r, q \mapsto s\}$$

This dependency rule indicates that the edge $r \mapsto s$ can be produced as a result of the assignment a whenever the edges $p \mapsto r$ and $q \mapsto s$ can be realized simultaneously.

The dependency rules can be created by examining a points-to graph G that results from a conservative analysis. Let us first consider assignments of the form $*^m p := *^n q$. For each such assignment, we generate a set of rules as follows:

- Let paths(p, m) denote the set of all paths of length m starting from p in G, and let paths$(q, n+1)$ be the set of all paths of length $n+1$ starting from q.
- Consider each pair of paths $\pi_1 \colon p \rightsquigarrow_m p' \in \text{paths}(p, m)$ and $\pi_2 \colon q \rightsquigarrow_{n+1} q' \in \text{paths}(q, n+1)$.
- We generate the dependency rule: $\left(p' \mapsto q' \xleftarrow{*^m p:=*^n q} E(\pi_1) \cup E(\pi_2)\right)$ where $E(\pi_i)$ denotes the edges in the path π_i for $i \in \{1, 2\}$.

The case for assignments of the form $*^m p := \&q$ is essentially the same, so we elide it here. Overall, we obtain the set of rules for a finite-memory problem by taking all such rules generated from all assignments $a \in A$.

Note that the time taken for rule generation and the number of rules generated can be shown to be a polynomial in the size of the problem and the number of edges in the points-to graph (which is in turn at most quadratic in the number of variables) [3]. The time taken is exponential in the number of dereferences in the pointer assignments, but usually this number is very small in practice (it is at most one for Andersen-style statements).

This rule generation can be done offline as described above to take advantage of an optimized, off-the-shelf points-to analysis, but it can also be performed online during the execution of Andersen's analysis. Consider a points-to edge e discovered in the course of Andersen's analysis while processing an assignment a. The edges traversed at this step to produce e are exactly the dependence edges needed to create an edge dependency rule (as in the rule construction algorithm described above).

Example 3. Figure 1 shows the edge dependency rules derived from the result of Andersen's Analysis for the problem in Example 1.

Witness Enumeration. Once edge dependency rules are generated, witness search is performed via witness *enumeration*, which constructs possible partial witnesses. Consider a rule $r \colon e \xleftarrow{a} E$. Rule r states that we can realize edge e

$$r \mapsto q \xleftarrow{r:=\&q} \emptyset \qquad x \mapsto g_1 \xleftarrow{x:=\&g_1} \emptyset \qquad y \mapsto g_1 \xleftarrow{y:=x} x \mapsto g_1$$

$$g_1 \mapsto q \xleftarrow{*x:=r} x \mapsto g_1, r \mapsto q \qquad g_1 \mapsto g_1 \xleftarrow{*x:=r} x \mapsto g_1, r \mapsto g_1$$

$$g_1 \mapsto g_1 \xleftarrow{*x:=y} x \mapsto g_1, y \mapsto g_1 \qquad r \mapsto g_1 \xleftarrow{r:=*x} x \mapsto g_1, g_1 \mapsto g_1$$

$$r \mapsto q \xleftarrow{r:=*x} x \mapsto g_1, g_1 \mapsto q \qquad p \mapsto g_1 \xleftarrow{p:=*r} r \mapsto g_1, g_1 \mapsto g_1 \qquad p \mapsto q \xleftarrow{p:=*r} r \mapsto g_1, g_1 \mapsto q$$

Fig. 1. The edge dependency rules for the problem in Example 1

via assignment a if we can realize the set of edges E simultaneously (i.e., in a state satisfying E, executing a creates the points-to edge e). Intuitively, we can realize the set E if we can find a chain of rules realizing each edge in E. Thus, enumeration proceeds by repeatedly rewriting edge sets based on dependency rules until reaching the empty set; the statements associated with the rules employed become the candidate witness (see [3] for a detailed definition).

Example 4. We describe a witness enumeration step for Example 1. Starting from the set $E\colon \{r \mapsto g_1, g_1 \mapsto g_1\}$ and using the rule $r\colon g_1 \mapsto g_1 \xleftarrow{*x:=y} \{x \mapsto g_1, y \mapsto g_1\}$, we can rewrite set E to a set E' as follows:

$$E\colon \{r \mapsto g_1, g_1 \mapsto g_1\} \quad \overset{r}{\rightharpoondown} \quad E'\colon \{x \mapsto g_1, y \mapsto g_1, r \mapsto g_1\}\,.$$

Often, we will write such transitions using the same format as the rule itself:

$$E\colon \{r \mapsto g_1, g_1 \mapsto g_1\} \quad \xleftarrow{*x:=y} \quad E'\colon \{x \mapsto g_1, y \mapsto g_1, r \mapsto g_1\}\,.$$

Not all rewriting steps lead to valid witnesses. In essence, we need to ensure that the witness search respects the concrete semantics of the statements. Recall the definition of realizability (Definition 1), which states that a set of edges E is realizable if it is a subset of edges in a realizable graph. A realizable graph must be an exact points-to graph. Therefore, we simply detect when the exactness constraint is violated, which we call a *conflict set*.

Definition 2 (Conflict Set). *A set of edges E is a* conflict set *iff there exist two or more outgoing edges $v \mapsto v_1, v \mapsto v_2 \in E$ for some vertex v.*

In addition to conflict detection, we guarantee termination in the finite-memory problem by stopping cyclic rewriting of edge sets. Intuitively, if we have $E_1 \overset{r_1}{\rightharpoondown} E_2 \overset{r_2}{\rightharpoondown} \ldots \overset{r_n}{\rightharpoondown} E_n$, wherein $E_n \supseteq E_1$, the corresponding statements have simply restored the points-to edges in E_1. Hence no progress has been made toward a complete witness. Since all cyclic rewriting is truncated, and we have a finite number of possible edge sets (since memory is finite), termination follows.

Performing witness enumeration with conflict set detection for each points-to fact derived by an initial analysis yields a precise flow-insensitive points-to analysis as captured by the theorem below. Proofs of all theorems are given in the appendix of our companion technical report [3].

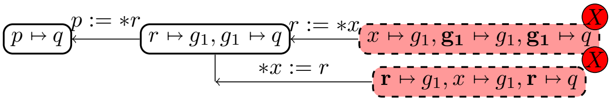
Theorem 1 (Realizability). *(A) An edge e is realizable iff there exists a sequence of rewrites w: $E_0 \colon \{e\} \xrightarrow{r_1} E_1 \xrightarrow{r_2} \cdots \xrightarrow{r_N} E_N \colon \emptyset$, such that none of the sets $E_0, \ldots, E_N$ are conflicting. (B) Furthermore, it is also possible to find w such that $E_i \not\supseteq E_j$ for all $i > j$.*

Example 5. Considering the problem from Example 1, the following sequence of rule applications demonstrates the realizability of the edge $r \mapsto g_1$:

$$\begin{aligned} \{r \mapsto g_1\} &\xleftarrow{r:=*x} \{x \mapsto g_1, g_1 \mapsto g_1\} \xleftarrow{*x:=y} \{x \mapsto g_1, y \mapsto g_1\} \\ &\xleftarrow{y:=x} \quad \{x \mapsto g_1\} \quad \xleftarrow{x:=\&g_1} \quad \emptyset\,. \end{aligned}$$

The sequence of assignments corresponding to the set of rule applications provides the witness sequence: $x := \&g_1; y := x; *x := y; r := *x;$.

The converse of Theorem 1 can be applied to show that a given edge is not realizable. To do so, we search over the sequence of applicable rules, stopping our search when a conflicting set or a superset of a previously encountered set of edges is encountered. A refutation tree for the non-realizability of edge $p \mapsto q$ from Example 1 is shown inset. In one path, the search terminates with a conflict on g_1, and in the other, the conflict is on r.



Possible Extensions. Looking beyond precise flow-insensitive points-to analysis, our algorithm can be extended to provide greater precision by introducing additional validation of the produced witnesses. For example, context sensitivity could be added by ensuring that each witness respects call-return semantics. One could add flow or even path sensitivity in a similar manner. This additional checking could be performed on partial witnesses during the search, possibly improving performance by reducing the size of the search space. Further study of these extensions is promising future work.

3.2 Handling Summarized Locations

In practice, problems arising from programming languages such as C will contain complications such as union types, structure types handled field insensitively, local variables in a recursive function, thread local variables, and dynamic memory allocations. Such constructs are often handled conservatively through *summary locations*, which model a (possibly unbounded) collection of concrete memory locations. As noted in Sect. 2, to conservatively model the potentially unbounded number of allocated cells with dynamic memory, Andersen's analysis uses one summary location per allocation site in the program.

The decidability of the precise flow-insensitive analysis in the presence of dynamic memory is unknown [5]. Here, we present two extensions to our algorithm that respectively handle summary locations in an over- and under-approximate manner, thereby yielding lower and upper bounds on the precision gap with a fully precise treatment of dynamic memory and other summarized locations.

Over-Approximating Summaries. To handle summary variables over-approximately, we can simply augment the search algorithm with weak update semantics for summaries. In particular, on application of a rule $r\colon e \overset{a}{\leftharpoondown} E$, if the source of edge e is a summary location, then e is not replaced in the rewriting (i.e., $E_0 \overset{r}{\rightharpoonup} E_0 \cup E$ for initial edge set E_0). Additionally, the definition of a conflict set (Definition 2) is modified to exclude the case when the conflict is on a summary location (i.e., two edges $v \mapsto v_1$ and $v \mapsto v_2$ where v is a summary location), as a summary may abstract an unbounded number of concrete cells. This handling clearly yields an over-approximate handling of summaries, as it is possible for the algorithm to generate witnesses that are not realizable by Definition 1. Hence, comparing Andersen's analysis and this algorithm yields a lower bound on the precision gap with a fully precise analysis.

Under-Approximating Summaries. To obtain an upper bound on the precision gap between Andersen's and the fully precise analysis, we define a straightforward under-approximating algorithm—during witness search, we treat summaries as if they were concrete memory locations. In essence, this approximation looks for witnesses that require only one instance of a summary (e.g., only one cell from a dynamic memory allocation site). This algorithm is unsound, as a points-to relation may be realizable even when this algorithm does not find a witness. However, if this algorithm finds a witness for a points-to relation, that relation is indeed realizable, and thus this algorithm yields an upper bound on the precision gap.

3.3 A Symbolic Encoding

In this section, we discuss a symbolic encoding for witness search and proving unrealizability. The idea is to encode the search for witnesses whose depths are bounded by some constant k using a Boolean formula $\varphi(e, k)$ such that any solution leads to a witness for edge e. We then adapt this search to infer the absence of witnesses by encoding subsumption checks. Crucially, our encoding allows *parallel updates* of unrelated pointer edges during witness search so that longer witnesses can be found at much smaller depths.

Propositions. For each edge $e \in E$ and depth $i \in [1, k+1]$, the Boolean variable $\mathsf{Edg}(e, i)$ denotes the presence of edge e in the set obtained at depth i. Similarly, for depths $i \in [1, k]$, the Boolean variable $\mathsf{Rl}(r, i)$ will be used to denote the application of the rule r at depth i (to obtain the set at depth $i + 1$). Note that there is no rule application at the last step.

Boolean Encoding. Some of the key assertions involved in the Boolean encoding are summarized in Table 1. The assertion $\mathsf{init}(e)$ describes the edge set at depth 1, which is required to be the singleton $\{e\}$. Recall that a pair of edges conflict if they have the same source location (which is not a summary location). The assertion $\mathsf{edgeConflict}(e_A, e_B, i)$ is used for such conflicting edges. Similarly, we

Table 1. Overview of the boolean encoding for witness search

Name	Definition	Remarks
$\mathsf{init}(e)$	$\mathsf{Edg}(e,1) \wedge \bigwedge_{e' \neq e} \neg\mathsf{Edg}(e',1)$	Start from edge set $\{e\}$
$\mathsf{edgeConflict}(e_A, e_B, i)$	$\neg\mathsf{Edg}(e_A, i) \vee \neg\mathsf{Edg}(e_B, i)$	Edges e_A, e_B cannot both be edges at depth i
$\mathsf{ruleConflict}(r_1, r_2, i)$	$\neg\mathsf{Rl}(r_1, i) \vee \neg\mathsf{Rl}(r_2, i)$	Rules r_1, r_2 cannot both be simultaneously applied at depth i
$\mathsf{someRule}(i)$	$\bigvee_{r \in R} \mathsf{Rl}(r, i)$	Some rule applies at depth i
$\mathsf{ruleApplicability}(r, i)$	$\mathsf{Rl}(r, i) \Rightarrow \mathsf{Edg}(e, i)$	Applying rule $r\colon e \leftarrow E$ at depth i creates edge e
$\mathsf{notSubsumes}(i, j)$	$\neg(\bigwedge_{e \in E} \mathsf{Edg}(e, i) \Rightarrow \mathsf{Edg}(e, j))$	Edge set at depth i does not contain set at depth j

define a notion of a conflict on the rules that enables parallel application of non-conflicting rules. Rules $r_1\colon e_1 \xleftarrow{a_1} E_1$ and $r_2\colon e_2 \xleftarrow{a_2} E_2$ are *conflicting* iff one of the following conditions holds: (a) $e_1 = e_2$, or (b) e_1 conflicts with some edge in E_2, or (c) e_2 conflicts with some edge in E_1. If two rules r_1, r_2 are not conflicting, then they may be applied in "parallel" at the same step and "serialized" arbitrarily, enabling the solver to find much longer witnesses at shallower depths. The corresponding assertion is $\mathsf{ruleConflict}(r_1, r_2, i)$. Assertion $\mathsf{someRule}(i)$ says some rule applies at depth i, and $\mathsf{ruleApplicability}(r, i)$ expresses the application of a rule r at depth i.

The assertion $\mathsf{ruleToEdge}(e, i)$ enforces that a rule $r\colon e \leftarrow E$ is applicable at depth i only if the corresponding edge e is present at that depth, which we define as follows (and is not shown in Table 1):

$$\mathsf{Edg}(e, i+1) \Leftrightarrow \left(\begin{array}{ll} (\mathsf{Edg}(e,i) \;\wedge\; \left(\bigwedge_{(r\colon e \leftarrow E) \in R} \neg\, \mathsf{Rl}(r,i)\right)) & \textit{/e existed previously/} \\ \quad \vee \bigvee_{(r'\colon e' \leftarrow E) \in R \text{ s.t. } e \in E} \mathsf{Rl}(r', i) & \textit{/or rule } r' \textit{ creates e/} \end{array} \right)$$

During the witness search, if we encounter an edge set E_i at depth i, such that $E_i \supseteq E_j$ for a smaller depth $j < i$, then the search can be stopped along that branch and a different set of rule applications should be explored. This aspect is captured by $\mathsf{notSubsumes}(i, j)$, and in the overall encoding below, we have such a clause for all depths $i > j$.

Overall Encoding. The overall encoding for an edge e_{query} is the conjunction:

$$\varphi(e_{\text{query}}, k)\colon \quad \bigwedge_{i \in [1,k]} \left[\begin{array}{ll} & \mathsf{init}(e_{\text{query}}) \\ \bigwedge_{e_1, e_2 \text{ conflicting}} & \mathsf{edgeConflict}(e_1, e_2, i) \\ \bigwedge_{r_1, r_2 \text{ conflicting}} & \mathsf{ruleConflict}(r_1, r_2, i) \\ \wedge & \mathsf{someRule}(i) \\ \wedge \bigwedge_{r \in R} & \mathsf{ruleApplicability}(r, i) \\ \wedge \bigwedge_{e \in E} & \mathsf{ruleToEdge}(e, i) \\ \wedge \bigwedge_{j \in [1, i-1]} & \mathsf{notSubsumes}(i, j) \end{array} \right] .$$

The overall witness search for edge e_{query}, consists of increasing the depth bound k incrementally until either (A) $\varphi(e_{\text{query}}, k)$ is unsatisfiable indicating a proof of unrealizability of the edge e_{query}, or (B) $\varphi(e_{\text{query}}, k) \ \wedge\ \mathsf{emptySet}(k+1)$ is satisfiable yielding a witness, wherein, the clause $\mathsf{emptySet}(i)\colon\ \bigwedge_{e \in E} \neg\mathsf{Edg}(e, i)$ encodes an empty set of edges.

Lemma 1. *(A) If $\varphi(e, k)$ is unsatisfiable then there cannot exist a witness for e for any depth $l \geq k$; and (B) If $\varphi(e, k) \ \wedge\ \mathsf{emptySet}(k+1)$ is satisfiable then there is a witness for the realizability of the edge e.*

4 Is There a Precision Gap in Practice?

We now describe our implementation of the ideas described thus far and the evaluation of these ideas to determine the size of the precision gap between Andersen's analysis and precise flow-insensitive analysis.

Implementation. Our implementation uses the C language front-end CIL [13] to generate a set of pointer analysis constraints for a given program. The constraint generator is currently field insensitive. Unions, structures, and dynamic memory allocation are handled with summary locations. To resolve function pointers, our constraint generator uses CIL's built-in Steensgaard analysis [18]. The constraints are then analyzed using our own implementation of Andersen's analysis. Our implementation uses a *semi-naive iteration* strategy to handle changes in the pointer graphs incrementally [14]. Other optimizations such as cycle detection have not been implemented, since our implementation of Andersen's analysis is not the scalability bottleneck for our experiments.

Our implementation of witness generation uses the symbolic witness search algorithm outlined in Sect. 3.3. Currently, our implementation uses the SMT solver Yices [6]. Note that the witness search directly handles statements with multiple dereferences from the original program, so the additional temporaries generated to run Andersen's analysis do not introduce imprecision in the search.

Evaluation Methodology. We performed our experiments over a benchmark suite consisting of 12 small- to medium-sized C benchmarks representing various Linux system utilities including network utilities, device drivers, a terminal application, and a system daemon. All measurements were taken on an 2.93 GHz Intel Xeon X7350 using 3 GB of memory.

To measure the precision gap for points-to analysis, we ran our witness search for all of the Andersen's points-to results for the benchmarks, both with over- and under-approximate handling of summary locations (yielding a lower and upper bound on the precision gap respectively, as described in Sect. 3.2). The primary result of this paper is that we found *no precision gap* between Andersen's analysis and the precise flow-insensitive analysis in either of these experimental configurations. In other words, our witness search never produced a refutation over our 12 benchmarks, no matter if summary locations were handled over- or under-approximately, and with precise handling of statements with multiple dereferences.

Following our observation that no precision gap exists for points-to queries, it is natural to consider if there is a precision gap between using Andersen's analysis to resolve alias queries and a precise flow-insensitive alias analysis. We say that p *aliases* q if there is a common location r such that both p and q may simultaneously point to r. We adapted the witness search encoding to search for witnesses for aliasing between pairs of variables that Andersen's analysis indicated were may-aliased. For aliasing experimental configurations, we ran the alias witness search for 1000 randomly chosen pairs of variables for each of our benchmarks (whereas for points-to configurations, we exhaustively performed witness search on all edges reported by Andersen's). Even though realizability of alias relations is more constrained than that of points-to relations, the search still produced a witness for all alias queries. This observation provides evidence that there is also likely no precision gap for alias analysis.

Results. As stated above, we found a flow-insensitive witness for *every points-to relation* and *every alias query* for our benchmarks in each experimental configuration. We found refutations for small hand-crafted examples that demonstrate the precision gap (like Examples 1 and 2), but not in real programs.

Table 2 gives details about the benchmarks and the execution of our witness-generating analyses. We show the statistics for two experimental configurations: the over- and under-approximating analyses for points-to queries with search over the original program statements.

Witness Search with Weak-Update Witnesses (Weak). For each points-to edge computed by Andersen's analysis, we performed a symbolic witness search using edge dependency rules derived with the original program statements until either a witness or a refutation for the edge was found. Weak-update semantics were used for summaries (see Sect. 3.2), yielding an over-approximate analysis and a lower bound on the precision gap.

Witness Search with Concretization (Conc). Here, we performed an under-approximate witness search that treated summaries as concrete locations, as described in Sect. 3.2. As refutations produced in this configuration may be invalid (due to the under-approximation), the configuration gives an upper bound on the precision gap.

The benchmarks are organized by function and sorted by number of lines of code in ascending order. The first set of columns gives statistics on the problem size, while the second set shows number of rules, analysis times, and search depths for each configuration. We note that running time depends primarily on the number of rules available to the witness search.

In Fig. 2, we show the per-benchmark distribution of discovered witness lengths for both the Weak configuration (left) and Conc configuration (right). Comparing each benchmark across the two configurations, we see relatively little change in the distribution. This result is a bit surprising, as one may expect that the more constraining Conc configuration would be forced to find longer witnesses. We hypothesize that the flow-insensitive abstraction allows so much flexibility in witness generation that there are always many possible witnesses

Table 2. Data from experiments using the Weak and Conc configurations. The "Program Size" columns give the number of thousands of lines of code (kloc), variables (vars), and pointer constraints (cons). Note that the number of variables includes all program variables (pointer type or non-pointer type), as any type may be used a pointer in C. The "Problem Size" columns give the number of rules generated (rules) and number of points-to edges found by Andersen's (edges). For the Weak and Conc experiments, we give the average search depth required and total running time.

	Program Size			Problem Size		Weak		Conc	
Benchmark	kloc	vars	cons	rules	edges	depth	time (s)	depth	time (s)
-NETWORK UTILITIES-									
aget (ag)	1.1	198	86	21	21	1.4	0.0	1.4	0.0
arp (ar)	3.1	1052	144	31	30	1.5	0.1	1.5	0.0
slattach (sl)	3.4	1046	164	31	31	1.5	0.1	1.5	0.0
netstat (ne)	4.5	1333	205	85	80	1.5	0.1	1.5	0.1
ifconfig (if)	8.8	1334	702	224	195	1.9	0.4	1.9	0.5
plip (pl)	18.4	4298	1556	167	146	2.5	1.0	2.7	1.2
-DEVICE DRIVERS-									
knot (kn)	1.3	243	125	22	21	1.7	0.0	1.7	0.0
esp (es)	10.9	3805	1475	6979	413	3.9	12937.0	4.2	734.0
ide-disk (id)	12.6	4684	1290	422	274	5.0	42.1	5.1	53.4
synclink (sy)	23.6	5221	2687	164	157	1.2	0.2	1.2	0.2
-TERMINAL APPLICATIONS-									
bc (bc)	6.2	658	615	1098	244	3.6	129.7	3.6	124.0
-DAEMONS-									
watchdog (wa)	9.4	1189	760	196	163	2.7	1.1	2.7	1.1

for each points-to relation regardless of whether we use the Weak or Conc configuration. The median witness length for Weak and Conc was 4, while the mean lengths were 8.41 and 8.58, respectively. Note that the mean lengths significantly exceeded the mean search depth for the benchmarks, indicating the effectiveness of parallel rule application in the search (see Sect. 3.3). The longest witness found in either configuration was of length 54.

4.1 Discussion: Why Is There No Precision Gap in Practice?

We note that the phenomena reported here as such defy a straightforward explanation that reflects directly on the way pointers are typically used in C programs.

From our experiments, we observe that not only does every edge discovered by Andersen's analysis have a witness, but it potentially has a *large number* of witnesses. This hypothesis is evidenced by the fact that we can (a) deploy non-conflicting rules in parallel and (b) discover long witnesses at a much smaller search depth. As a result, each witness consists of *parallel threads* of unrelated pointer assignments that contribute towards the final goal but can themselves be *interleaved* in numerous ways.

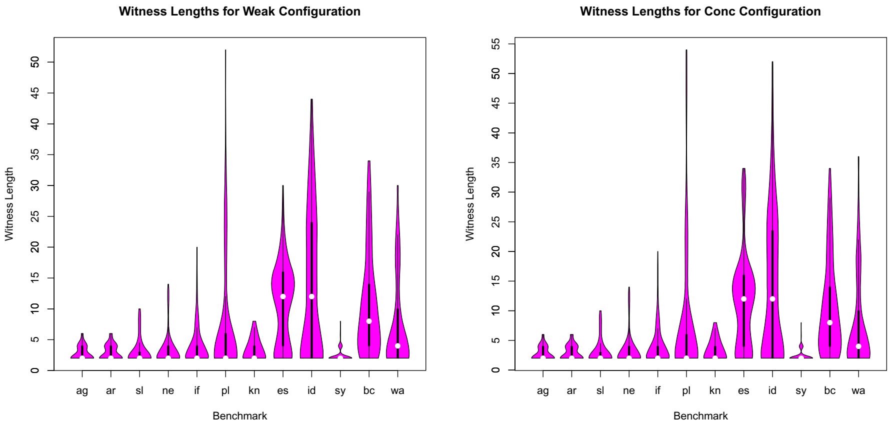


Fig. 2. Distribution of witness lengths for the WEAK and CONC configurations. The white dot shows the median length, the endpoints of the thick line give the first and last quartiles, and the thin line indicates the first and last deciles. The width of the plot indicates the (relative) density of witnesses for a given length.

Recall that the rules obtained from Andersen's analysis are of the form $e \xleftarrow{a} \{e_1, e_2\}$, stating that if e_1, e_2 are simultaneously realizable then e is realizable by application of assignment a. Therefore, unrealizability of e means that for every such rule that can realize e, the corresponding RHS set $\{e_1, e_2\}$ are simultaneously unrealizable. In turn, this indicates that any sequence of assignments that realizes e_1 destroys e_2 and vice versa. Such "mutually-destructive" pairs of points-to relations are easy to create and maintain in programs. However, these examples depend on sequential control flow to produce the desired behavior. When analyzed under flow-insensitive semantics wherein statements can occur multiple times under varying contexts, the behavior changes drastically.

Other examples of imprecisions in points-to analysis depend on the proper modeling of function calls and returns. For example, the following code may be used to initialize a linked list:

```
void initList(List* l) { l->header->next = l->header->prev = l->header; }
```

If this function were invoked at multiple contexts with different arguments, Andersen's analysis could conflate the internal list pointers while a precise flow-insensitive analysis would not (assuming a context-insensitive treatment of the function). However, note that this precision gain would require the ability to distinguish the list nodes themselves, for which flow-insensitive analysis is often insufficient. Furthermore, the precision gain would be quite fragile; if the above source is rewritten to store `l->header` in a temporary variable, the gain disappears. Stepping out of pure flow-insensitive analysis, a partially-flow-sensitive analysis [17] would be more robust to such changes and may be worth future investigation.

4.2 Threats to Validity

One threat to the validity of our results is that they may be sensitive to how various C language constructs are modeled by our constraint generator. It is possible that field sensitivity, (partial) context sensitivity, or a more precise treatment of function pointers would expose a precision gap. However, given the exacting conditions required for a gap to arise, we believe it is unlikely that these other axes of precision would affect our results in any significant way.

It is also possible that our benchmarks are not representative of small- to medium-sized C programs. To mitigate this concern, we chose benchmarks from several domains: network utilities, device drivers, a command-line application, and a system daemon. We also attempted to select programs of different sizes within the spectrum of small- to medium sized programs. Although no benchmark suite can be representative of all programs, our intent was to choose a reasonable number of programs with diverse sizes and uses to comprise a set that adequately represents small- to medium-sized C programs.

Finally, it may be that the precision gap only manifests itself on larger programs than the ones we considered. We have tried to perform measurements on examples in the 25 to 200 kloc range, but such examples are presently beyond the reach of our implementation. We are currently investigating implementing ideas along the lines of bootstrapping [10], wherein the witness search may focus on a smaller subset of edges in the points-to graph and allow our experiments to scale to larger programs. Despite our inability to scale to programs beyond 25k lines, we hypothesize that our conclusion generalizes to larger programs based on the intuitions outlined in Sect. 4.1.

5 Related Work

Our work was partially inspired by previous work on the complexity of precise points-to analysis variants. Horwitz [9] discussed the precision gap between Andersen's analysis and precise flow-insensitive analysis and proved the NP-hardness of the precise problem. Chakaravarthy [5] gave a polynomial-time algorithm for precise flow-insensitive analysis for programs with well-defined types.

Muth and Debray [12] provide an algorithm for a variant of precise flow-sensitive points-to analysis (for programs without dynamic memory) that can be viewed as producing witnesses by enumerating all possible assignment sequences and storing the exact points-to graph, yielding a proof of PSPACE-completeness. Others have studied the complexity and decidability of precise flow-sensitive and partially-flow-sensitive points-to analysis [11, 15, 17].

The edge reduction rules derived in our approach are similar, in spirit, to the reduction from pointer analysis problems to graph reachability as proposed by Reps [16]. However, a derivation in this CFL for a points-to edge need not always yield a witness. In analogy with Andersen's analysis, the derivation may ignore conflicts in the intermediate configurations. Finding a derivation in a CFL without conflicting intermediate configurations reduces to temporal model checking of push-down systems. This observation, however, does not seem to yield a better complexity bound [4].

Our work employs SAT solvers to perform a symbolic search for witnesses to points-to edges. Symbolic pointer analysis using BDDs have been shown to outperform explicit techniques in some cases by promoting better sharing of information [2, 19].

6 Conclusion

We have presented techniques for measuring the precision gap between Andersen's analysis and precise flow-insensitive points-to analysis in practice. Our approach is based on refinement of points-to analysis results with a witness search and a symbolic encoding to perform the search with a tuned SAT solver. Our experimental evaluation showed that for medium-sized C programs, the precision gap between Andersen's and precise flow-insensitive analysis is (as far as we can observe) non-existent. Future work includes improving the scalability of our witness search algorithm and applying our techniques to other languages. We also plan to extend the witness search algorithm to incorporate higher levels of precision, including context sensitivity and some form of flow sensitivity.

Acknowledgments. We thank Jeffrey S. Foster for fruitful discussions on an earlier draft of this paper, as well as the anonymous reviewers for their helpful comments. The authors are also grateful to Gogul Balakrishnan, Franjo Ivancic, and Aarti Gupta at NEC Laboratories America in Princeton, NJ for helping us with the Linux device driver benchmarks used in our experiments. We also thank Jan Wen Voung, Ranjit Jhala, and Sorin Lerner for including a large set of C benchmarks in their publicly available Relay/Radar tool. This research was supported in part by NSF under grants CCF-0939991 and CCF-1055066.

References

[1] Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. PhD thesis, University of Copenhagen, DIKU (1994)

[2] Berndl, M., Lhoták, O., Qian, F., Hendren, L., Umanee, N.: Points-to analysis using BDDs. In: Programming Language Design and Implementation (PLDI), pp. 103–114 (2003)

[3] Blackshear, S., Chang, B.-Y.E., Sankaranarayanan, S., Sridharan, M.: The flow-insensitive precision of Andersen's analysis in practice (extended version). Technical Report CU-CS-1083-11, Department of Computer Science, University of Colorado Boulder (2011)

[4] Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Application to model-checking. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 135–150. Springer, Heidelberg (1997)

[5] Chakaravarthy, V.T.: New results on the computability and complexity of points-to analysis. In: Principles of Programming Languages (POPL), pp. 115–125 (2003)

[6] Dutertre, B., de Moura, L.: The YICES SMT solver, http://yices.csl.sri.com/tool-paper.pdf

[7] Hardekopf, B., Lin, C.: The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In: Programming Language Design and Implementation (PLDI), pp. 290–299 (2007)

[8] Hind, M.: Pointer analysis: Haven't we solved this problem yet? In: Program Analysis for Software Tools and Engineering (PASTE), pp. 54–61 (2001)

[9] Horwitz, S.: Precise flow-insensitive alias analysis is NP-hard. ACM Trans. Program. Lang. Syst. 19(1) (1997)

[10] Kahlon, V.: Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In: Programming Language Design and Implementation (PLDI), pp. 249–259 (2008)

[11] Landi, W.: Undecidability of static analysis. ACM Lett. Program. Lang. Syst. 1(4), 323–337 (1992)

[12] Muth, R., Debray, S.: On the complexity of flow-sensitive dataflow analyses. In: Principles of Programming Languages (POPL), pp. 67–80 (2000)

[13] Necula, G., McPeak, S., Rahul, S., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: CC 2002. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002)

[14] Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, Heidelberg (1999)

[15] Ramalingam, G.: The undecidability of aliasing. ACM Trans. Program. Lang. Syst. 16(5), 1467–1471 (1994)

[16] Reps, T.: Program analysis via graph reachability. Information and Software Technology 40, 5–19 (1998)

[17] Rinetzky, N., Ramalingam, G., Sagiv, M., Yahav, E.: On the complexity of partially-flow-sensitive alias analysis. ACM Trans. Program. Lang. Syst. 30(3) (2008)

[18] Steensgaard, B.: Points-to analysis in almost linear time. In: Principles of Programming Languages (POPL), pp. 32–41 (1996)

[19] Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: Programming Language Design and Implementation (PLDI), pp. 131–144 (2004)

Side-Effect Analysis of Assembly Code

Andrea Flexeder, Michael Petter, and Helmut Seidl

Technische Universität München, Boltzmannstrasse 3, 85748 Garching, Germany
{flexeder,seidl,petter}@cs.tum.edu
http://www2.cs.tum.edu/~{flexeder,seidl,petter}

Abstract. In this paper we present a light-weight interprocedural side-effect analysis on assembly code. We represent the modifying potential of a procedure f by classifying all write accesses, occurring within f, relative to the parameter registers. In particular our approach is the first to accurately handle reference parameters. We demonstrate the usefulness of this approach by integrating this analysis into our assembly analyser and provide an evaluation of the precision of our approach. Approximately 50 per cent of all procedures can be statically shown to have side-effects.

1 Introduction

The prevalent binary analysis frameworks, e.g. CodeSurfer/$x86$ [23], BitBlaze [28] or Jakstab [16] reach their limits when analysing large assemblies. All of these approaches rely on the (approximate) call-string approach [27] and thus suffer from imprecision when limiting call-string length. Analysing binaries consisting of more than $700,000$ assembler instructions with these call-string based approaches will be prohibitively expensive to be applied in practise. On the one hand, analyses can treat procedure calls conservatively, i.e. a procedure call invalidates the values of *all* the local variables whenever a pointer to its stack frame escapes to another execution context. On the other hand, analyses can make standard assumptions about a procedure call, e.g. that a procedure does only write to its own stack frame or to the heap. While these assumptions hold for restricted program classes, e.g. safety-critical software, this is not the case for general-purpose software as our experiments show. Therefore, we propose a *side-effect analysis* which computes a tight approximation of write accesses of procedures to the stack. In contrast to the existing binary analysis frameworks, we propose a light-weight form of functional approach [27] to interprocedural analysis which still produces useful results. In our framework, the effect of a procedure is represented by all possible parameter register relative write accesses. The following example illustrates the side-effect of a procedure which is passed a local address.

Example 1. In the following C code fragment, a pointer to array `c` is passed as parameter to procedure `f`. Within `f`, the first four elements of the array are set to zero. The corresponding PPC [14] assembly is given to the right. In PPC assembler the stack pointer is given by register `r1`. The array access is realised by passing a stack address, i.e., address `r1+12`, in register `r3` to procedure `f` (cf. instruction `0x5C`). After the call to `f` within `main`, a conservative analysis of `main` therefore must assume that all local

E. Yahav (Ed.): SAS 2011, LNCS 6887, pp. 77–94, 2011.

variables of `main` may have been modified by the procedure call. Thus, all information about the values of local variables after the procedure call `f()` is lost.

```
f(int a[]){
  int j;
  for(j=0;j<4;j++)
    a[j] = 0;
}

main(){
  int c[4];
  int b = 13;
  f(c);
}
```

```
//main(){...
//int b = 13;
50:  li    r2,13
54:  stw   r2,8(r1)
//f(c);
58:  addi  r2,r1,12
5C:  mr    r3,r2
60:  bl    0x00
//}...
```

```
//f(int a[]){
00:  stwu    r1,-32(r1)
04:  stw     r3,24(r1)
//...

//a[j] = 0;
14:  lwz     r2,8(r1)
18:  mulli   r2,r2,4
1C:  mr      r9,r2
20:  lwz     r2,24(r1)
24:  add     r9,r9,r2
28:  li      r2,0
2C:  stw     r2,0(r9)
//...
```

In order to retain the value of the local variable `b` in `main`, we have to determine the side-effect of `f`. The side-effect identifies all stack locations whose value may be modified through the procedure call and thus must be invalidated after instruction `0x60`. Accordingly, we inspect the memory write accesses occurring in procedure `f`. The only memory write access in `f` happens at instruction `0x2C` and modifies the memory cells addressed by the expressions `{r3,r3+4,r3+8,r3+12}`. This set describes the *modifying potential* of procedure `f`. By matching actual to formal parameters, we conclude that within procedure `main` after the call to `f`, the memory locations `{r1+12,r1+16,r1+20,r1+24}` of `main` are modified, while local `b` (i.e., the memory cell with address `r1+8`) remains untouched by `f()`. □

Related Work. Side-effect analysis has intensively been studied for high-level languages, e.g., the purity analysis for Java programs [25], the side-effect analysis for C/C++ programs [4,18], or the context-sensitive pointer analysis for C programs [31,11]. The approach of Cooper et al. [5] relies on graph algorithms for solving the alias-free side-effect analysis problem introduced by Banning [3] in a flow-insensitive manner. All these techniques, however, are not directly applicable to low-level code where there is no clear distinction between an integer and an address.

For the analysis of low-level code, Debray et al. present an interprocedural, flow-sensitive pointer alias analysis of $x86$ executables, which, however, is context-insensitive [8]. They abstract the value of each register by an *address descriptor*, i.e., a set of possible congruence values with respect to a program instruction. They make no distinction between two addresses which have the same lower-order k bits. Since they do not track any memory content they suffer from information loss, whenever a register is assigned a value from memory. Moreover if different definitions of the same register reach the same join point, then this register is assumed to take any value. A context-sensitive low-level points-to analysis is presented in [15]. This approach is only partially flow-sensitive: the values of registers are handled flow-sensitively using SSA form, while memory locations are treated flow-insensitively (tracking only a single points-to set for each memory location). The notion of *UIV*s (unknown initial values) is introduced in order to represent all those memory locations that are accessible by the procedure but do not belong to the current stack frame of the procedure or to the stack frames of its callees. Their aim is to improve on compiler optimisations, such as e.g. load and store reorderings, and thus a crude treatment of local memory is sufficient. In contrast, we track local memory locations context-sensitively as well. The most advanced approach in the area of analysis of assembly code is provided by Reps et al. [23]. They propose a flow-sensitive framework for analysing memory accesses in $x86$ executables. For dealing with procedures, they rely on the call-string approach (CSA)[27]. In order overcome the context-insensitivity of CSA-0 and the impracticability of increasing the length of call-strings they apply techniques from Cooper et al. [5] in order to determine the set of memory locations that may be modified by each procedure. This information is used in order to improve on the precision when combining the information from different call sites. In contrast, our algorithm adheres to the functional approach of program analysis and thus does not suffer from the limitations of call-strings of bounded length.

A stack analysis for $x86$ executables is addressed in [19]. There, the authors aim at verifying that a function leaves the stack in its original state after the function returns. In order to identify such *well-behaving* functions, use-depth and kill-depth analyses are introduced. By means of use-depth information they estimate the maximal stack height from which a function may read a value, while kill-depth information tells the maximal height of the runtime stack to which the function may write a value. While addressing a related problem, the applied techniques are different. In particular, our approach does not suffer from a loss of precision when dealing with recursive procedures.

Contributions. We propose a light-weight side-effect analysis based on the functional approach to interprocedural analysis which

- provides context-sensitive side-effects by computing an overapproximation of the set of register-relative memory locations which are modified by a procedure call.
- allows analysing large binaries with more than $700,000$ assembler instructions reasonably fast and also provides significant side-effects.
- can be used to lose less information after procedure calls in sound intraprocedural analyses.

Overview. In the next Section 2 we present the concrete semantics for our side-effect analysis. In Section 3 we first describe how to embed the side-effect of a procedure into an intraprocedural analysis, then present how to compute the modifying potential of a procedure and finally provide a correctness proof of our approach. Additionally we describe the implementation of the side-effect analysis in our assembly analyser and evaluate its results in Section 4, before we conclude in Section 5.

2 The Concrete Semantics

For the analysis, we assume that programs are given as a collection of procedures q where each q is represented by a finite control flow graph G_q. The control flow graph of a given assembly can be extracted via methods provided by, e.g., the program analysis frameworks [12,24,17]. Every control flow graph G_q consists of:

- a finite set N_q of *program points* of the procedure q,
- a finite set $E_q \subseteq (N_q \times \mathit{Label} \times N_q)$ of *edges*, where *Label* denotes a program instruction,
- a unique entry point $s_q \in N_q$ for procedure q,
- a unique exit point $r_q \in N_q$ for procedure q and
- a designated procedure $main$ where program execution always starts.

Let $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$ denote the set of registers the assembly operates on, with k the number of registers. We assume that all the registers $\mathbf{X}$ are global and thus possibly serve for passing parameters.

The concrete effect of every processor instruction within our analysis is modelled by a sequence of the following constructs:

- stm: assignment statements and memory access instructions,
- guards and
- $q()$: procedure calls.

Each edge in the control flow graph therefore is annotated by one of these constructs. For the analysis, we consider assignments of the form $\mathbf{x}_i := t$ ($\mathbf{x}_i$ a register, t an expression) and guards providing comparisons of the values of two registers, i.e.,$\mathbf{x}_j \bowtie \mathbf{x}_k$, and the value of a register with a constant, i.e. $\mathbf{x}_j \bowtie c$, for every comparison operator $\bowtie$. Furthermore, we use non-deterministic assignments of the form $\mathbf{x}_i :=?$ to represent instructions possibly modifying register $\mathbf{x}_i$ in an unknown way. Non-deterministic assignments represent a safe description of those instructions which are not handled precisely, but have an impact on the values of registers, as for instance bit operations. Memory access instructions are of the form $\mathbf{x}_i := M[t]$ or $M[t] := \mathbf{x}_i$ ($\mathbf{x}_i$ a register, t an expression). For simplicity, here we only consider memory access instructions applied to words, i.e., 4 bytes of memory simultaneously. That means that we exclude instructions operating on multiple registers simultaneously. The formalism can be extended, though, to deal with variable-length memory access instructions and multi-register arithmetic as well. This format subsumes assembler programs in three-address form as is provided by some processors, e.g., PowerPC[14], or by a transformation into a low-level intermediate representation, e.g., REIL [10].

When dealing with procedure calls, we distinguish procedure-global from procedure-local memory locations. A dedicated register of the processor serves as *stack pointer*, i.e., holds the top-of-stack for the current procedure. In our formalisation, this register is given by $\mathbf{x}_1$. Accordingly, the single edge starting at the entry point of the control flow graph for a procedure q is assumed to be annotated with an instruction $\mathbf{x}_1 := \mathbf{x}_1 - c_q$ for some constant c_q. This instruction allocates the local stack space for the current invocation of q. Likewise, the single edge reaching the exit point of the control flow graph for q is annotated with the instruction $\mathbf{x}_1 := \mathbf{x}_1 + c_q$ for the same constant c_q. This instruction deallocates the local stack space. For simplicity, we rule out intermediate increments or decrements of the stack pointer, and thus do not consider *dynamic* stack allocation here. An approach to deal with dynamic stack allocation is sketched in [13]. Figure 1 illustrates our control flow representation of Example 1.

For the concrete semantics, the memory $\mathbf{S}$ of the program is divided into the disjoint addresses spaces $\mathbf{S}_L$ for the *stack* or local memory, and $\mathbf{S}_G$ for global memory. The set

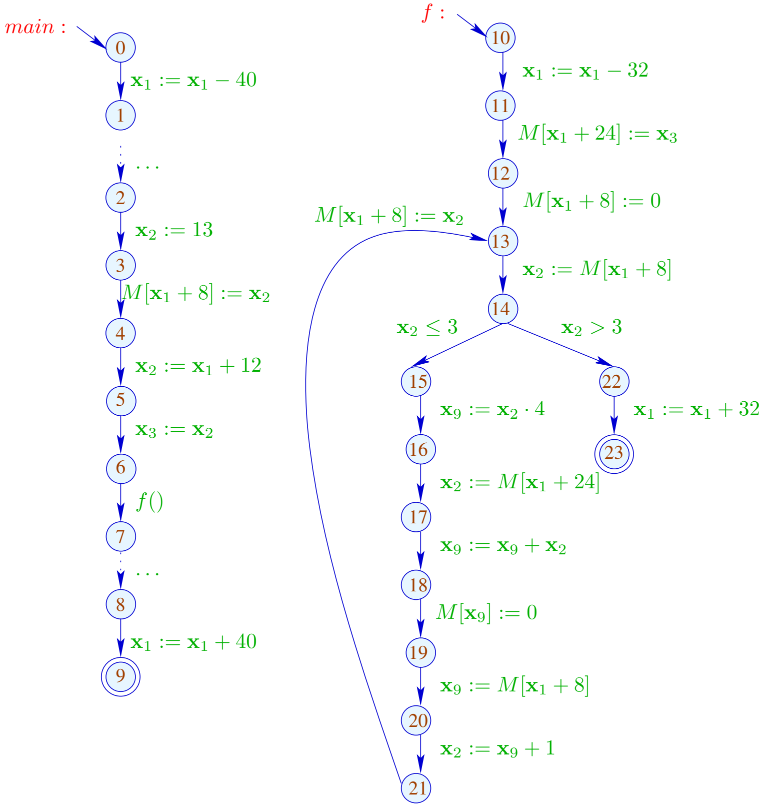

$main$:
0
$\mathbf{x}_1 := \mathbf{x}_1 - 40$
1
...
2
$\mathbf{x}_2 := 13$
3
$M[\mathbf{x}_1 + 8] := \mathbf{x}_2$
4
$\mathbf{x}_2 := \mathbf{x}_1 + 12$
5
$\mathbf{x}_3 := \mathbf{x}_2$
6
$f()$
7
...
8
$\mathbf{x}_1 := \mathbf{x}_1 + 40$
9
f :
10
$\mathbf{x}_1 := \mathbf{x}_1 - 32$
11
$M[\mathbf{x}_1 + 24] := \mathbf{x}_3$
12
$M[\mathbf{x}_1 + 8] := 0$
$M[\mathbf{x}_1 + 8] := \mathbf{x}_2$
13
$\mathbf{x}_2 := M[\mathbf{x}_1 + 8]$
14
$\mathbf{x}_2 \leq 3$
$\mathbf{x}_2 > 3$
15
22
$\mathbf{x}_9 := \mathbf{x}_2 \cdot 4$
$\mathbf{x}_1 := \mathbf{x}_1 + 32$
16
23
$\mathbf{x}_2 := M[\mathbf{x}_1 + 24]$
17
$\mathbf{x}_9 := \mathbf{x}_9 + \mathbf{x}_2$
18
$M[\mathbf{x}_9] := 0$
19
$\mathbf{x}_9 := M[\mathbf{x}_1 + 8]$
20
$\mathbf{x}_2 := \mathbf{x}_9 + 1$
21

Fig. 1. Control flow representation of Example 1

of global addresses is given by $\mathbf{S}_G = \{(G, z) \mid z \in \mathbb{Z}\}$, and the set of stack addresses is given by $\mathbf{S}_L = \{(L, z) \mid z \in \mathbb{Z}\}$. A program state γ then is given by a triple $\gamma = \langle \rho, f, \lambda \rangle$ where

- $\rho : \mathbf{X} \to \mathbf{S}$ provides the contents of registers.
- $f = (x_r, x_{r-1}, \ldots, x_0)$ for $x_r < x_{r-1} < \ldots < x_0$ is the *frame structure* of the current stack where $x_0 = \mathsf{Max} + 1$ and x_r equals the least address on the current stack. Here Max is the maximal height the stack can grow. Thus in particular, $x_r = \rho(\mathbf{x}_1)$. Note that here the stack grows *downward* from numerically higher addresses towards zero.
- λ provides the contents of memory locations. Thus, λ assigns values to the set $\mathbf{S}_G$ of global addresses as well as to the current set of local addresses. The values of global addresses are restricted to be global addresses only. The set of allowed local addresses is restricted to the set $\mathbf{S}_L(x_r) = \{(L, z) \mid x_r \leq z < x_0\}$. Their values are restricted to elements from the set $\mathbf{S}_G \cup \mathbf{S}_L(x_r)$ only.

In order to obtain a uniform representation, concrete integer values are embedded into the *global* address space, i.e., the plain value 5 as well as the global address 5 are represented by the pair $(G, 5)$. Remark that the labels L and G allow to inherently distinguish between local and global address space.

We consider address arithmetic w.r.t. a given frame structure $f = (x_r, x_{r-1}, \ldots, x_0)$. We restrict the arithmetic operations on local addresses such that starting from a local address from a stack frame $[x_i, x_{i-1} - 1]$, arithmetic is only legal if it produces a local address within the range $[x_i, x_0]$. The interval $[x_i, x_0]$ represents the set of all currently valid local addresses. Accordingly, the arithmetic operations of addition, subtraction, multiplication and boolean comparisons on elements from $\{L, G\} \times \mathbb{Z}$ w.r.t. a given frame structure $f = (x_r, x_{r-1}, \ldots, x_0)$ are defined by:

$$
\begin{array}{l}
(L, c_1) +_f (L, c_2) = \textsf{undefined} \\
(G, c_1) +_f (G, c_2) = (G, c_1 + c_2) \\
(L, c_1) +_f (G, c_2) = (G, c_2) +_f (L, c_1) = \begin{cases} (L, c_1 + c_2) & \text{if } x_r \leq c_1 \Rightarrow x_r \leq c_1 + c_2 \\ \textsf{undefined} & \text{otherwise} \end{cases} \\
(L, c_1) -_f (L, c_2) = (G, c_1 - c_2) \\
(G, c_1) -_f (G, c_2) = (G, c_1 - c_2) \\
(L, c_1) -_f (G, c_2) = \begin{cases} (L, c_1 - c_2) & \text{if } x_r \leq c_1 \implies x_r \leq c_1 - c_2 \\ \textsf{undefined} & \text{otherwise} \end{cases} \\
(G, c_1) -_f (L, c_2) = \textsf{undefined} \\
(G, c_1) \cdot_f (G, c_2) = (G, c_1 \cdot c_2) \\
(L, c_1) \cdot_f (G, c_2) = (G, c_2) \cdot_f (L, c_1) = \textsf{undefined} \\
(L, c_1) \cdot_f (L, c_2) = \textsf{undefined} \\
(L, c_1) \bowtie (L, c_2) \begin{cases} \textsf{true} & \text{if } c_1 \bowtie c_2 \\ \textsf{false} & \text{otherwise} \end{cases} \\
(G, c_1) \bowtie (G, c_2) \begin{cases} \textsf{true} & \text{if } c_1 \bowtie c_2 \\ \textsf{false} & \text{otherwise} \end{cases} \\
(G, c_1) \bowtie (L, c_2) = \textsf{undefined} \\
(L, c_2) \bowtie (G, c_1) = \textsf{undefined}
\end{array}
$$

for every comparison operator $\bowtie$. If an $\textsf{undefined}$ value occurs, we assume that an exception is thrown and the program execution is aborted. For the analysis, we only consider non-aborting program executions and flag warnings if an abortion cannot be excluded. Statements and guards induce transformations of (sets of) states. A single processor instruction s on a given program state $\langle \rho, f, \lambda \rangle$ returns a set of program states which is defined by:

$$
\begin{array}{lcl}
[\![\mathbf{x}_i := t]\!]\langle \rho, f, \lambda \rangle & = & \{\langle \rho \oplus \{\mathbf{x}_i \mapsto [\![t]\!](\rho, f)\}, f, \lambda \rangle\} \\
[\![\mathbf{x}_i :=?]\!]\langle \rho, f, \lambda \rangle & = & \{\langle \rho \oplus \{\mathbf{x}_i \mapsto a\}, f, \lambda \rangle \mid a \in \mathbf{S}_G\} \\
[\![\mathbf{x}_i := M[t]]\!]\langle \rho, f, \lambda \rangle & = & \{\langle \rho \oplus \{\mathbf{x}_i \mapsto \lambda([\![t]\!](\rho, f))\}, f, \lambda \rangle\} \\
[\![M[t] := \mathbf{x}_i]\!]\langle \rho, f, \lambda \rangle & = & \{\langle \rho, f, \lambda \oplus \{\lambda([\![t]\!](\rho, f)) \mapsto \rho(\mathbf{x}_i)\}\rangle\} \\
[\![\mathbf{x}_j \bowtie \mathbf{x}_k]\!]\langle \rho, f, \lambda \rangle & = & \{\langle \rho, f, \lambda \rangle \mid \rho(\mathbf{x}_j) \bowtie \rho(\mathbf{x}_k) = \textsf{true}\} \\
[\![\mathbf{x}_j \bowtie c]\!]\langle \rho, f, \lambda \rangle & = & \{\langle \rho, f, \lambda \rangle \mid \rho(\mathbf{x}_j) \bowtie (G, c) = \textsf{true}\}
\end{array}
$$

with $\mathbf{x}_i, \mathbf{x}_j, \mathbf{x}_k \in \mathbf{X}$ where $i \neq 1$, t an expression and arbitrary $c \in \mathbb{Z}$. Here, the operator $\oplus$ adds new argument/value pairs to a function. Moreover, the evaluation function $[\![t]\!](\rho, f)$ takes an expression t and returns the value of t in the context of register assignment ρ w.r.t. a given frame structure f.

$$[\![t]\!](\rho, f) = \begin{cases} [\![t_1]\!](\rho, f) \,\Box_f\, [\![t_2]\!](\rho, f) & \text{if } t = t_1 \,\Box\, t_2 \\ \rho(\mathbf{x}_i) & \text{if } t = \mathbf{x}_i \\ (G, c) & \text{if } t = c \end{cases}$$

with $\Box = \{+, -, \cdot\}$ and $\Box_f = \{+_f, -_f, \cdot_f\}$, respectively. Consider, e.g., a memory access $\mathbf{x}_i := M[t]$. Then for every state $\langle \rho, f, \lambda \rangle$ the value a of expression t is determined. Given that a is a valid memory address of λ, the content $\lambda(a)$ of the memory location a is assigned to register $\mathbf{x}_i$.

Next, we describe the effect of a procedure call $q()$. According to our convention, a stack pointer decrement instruction reserves a stack region for the local variables of the procedure. For the concrete semantics, this means that for a procedure call $q()$ the stack is extended by the stack frame of q. According to our assumptions, the only assignments to variable $\mathbf{x}_1$ are of the form $\mathbf{x}_1 := \mathbf{x}_1 + c$ for some $c \in \mathbb{Z}$ at the first and last control flow edge of control flow graphs only. There, these assignments allocate or deallocate the local stack frame for the current instance of the procedure. The newly allocated memory cells are uninitialised and therefore have arbitrary values. Accordingly, we define for $c > 0$ and $f = (x_r, \ldots, x_0)$:

$$\begin{aligned} [\![\mathbf{x}_1 := \mathbf{x}_1 - c]\!]\langle \rho, f, \lambda \rangle = \{ & \langle \rho \oplus \{\mathbf{x}_1 \mapsto \rho(\mathbf{x}_1) -_f (G, c)\},\ (x_r - c, x_r, \ldots, x_0), \\ & \lambda \oplus \{(L, z) \mapsto a \mid z \in [x_r - c, x_r); a \in \mathbf{S}_G\}\rangle \mid x_r \geq c\} \end{aligned}$$

After execution of the body of the procedure, the current stack frame is deallocated. Assume that $f = (x_{r+1}, x_r, \ldots, x_0)$ and $c > 0$. Then the effect of the last instruction of the procedure epilogue which increments the stack pointer again, for state $\langle \rho, f, \lambda \rangle$ is given by:

$$\begin{aligned} [\![\mathbf{x}_1 := \mathbf{x}_1 + c]\!]\langle \rho, f, \lambda \rangle = \{ & \langle \rho \oplus \{\mathbf{x}_1 \mapsto \rho(\mathbf{x}_1) +_f (G, c)\}, (x_r, \ldots, x_0), \lambda_{|\mathbf{S}_G \cup \mathbf{S}_{L(x_r)}} \rangle \\ & \mid x_r = x_{r+1} + c\} \end{aligned}$$

After executing the last instruction of procedure q, the stack frame of q has been popped. In particular, the stack pointer again points to the top of the stack. With $\lambda_{|M}$ we denote the restriction of λ to the domain M.

Any procedure q transforms the set of program states before the procedure call to the set of program states after the procedure call to q. Following the approach, e.g., of [21,22], we characterise the effect of a procedure p by means of a constraint system E over transformers operating on program states Γ:

$$\begin{array}{lll} E(s_p) & \supseteq \mathsf{Id} & s_p \text{ start point of procedure } p \\ E(v) & \supseteq E(r_q) \circ E(u) & (u, q(), v) \text{ a call edge} \\ E(v) & \supseteq [\![s]\!] \circ E(u) & (u, s, v) \text{ with } s \in \mathsf{stm} \cup \mathsf{guards} \end{array}$$

with r_q the return point of procedure q. Here, Id denotes the singleton mapping defined by $\mathsf{Id}(y) = \{y\}$ and $\circ$ denotes the composition of transformations of type $\Gamma \rightarrow 2^{\Gamma}$.

This composition is defined by:

$$(f \circ g)(y) = \bigcup \{f(y') \mid y' \in g(y)\}.$$

The effect of a procedure q is given by the effect accumulated at its return point $E(r_q)$.

The set of *attained program states* when reaching program point u is given by the least solution of the following system of inequations R. The effect of a call to procedure q is given by the application of $E(r_q)$ to the set of program states, valid immediately before the procedure call.

$$\begin{array}{lll} R(s_{main}) \supseteq \Gamma_0 & & \\ R(v) \quad \supseteq [\![s]\!]\,(R(u)) & & (u, s, v) \text{ with } s \in \mathsf{stm} \cup \mathsf{guards} \\ R(v) \quad \supseteq E(r_q)(R(u)) & & (u, q(), v) \text{ a call edge} \end{array}$$

Here, Γ_0 denotes the set of start states of the program. At program start, the stack is empty. This means that the stack pointer $\mathbf{x}_1$ has the value $(L, \mathsf{Max} + 1)$. Thus,

$$\Gamma_0 = \{\langle \rho, (\mathsf{Max} + 1), \lambda\rangle \mid \rho : \mathbf{X} \to \mathbf{S}_G \mid \rho(\mathbf{x}_1) = (L, \mathsf{Max} + 1), \lambda : \mathbf{S}_G \to \mathbf{S}_G\}$$

Moreover, *application* of a function $T : \Gamma \to 2^{\Gamma}$ to a set $Y \subseteq \Gamma$ is defined by:

$$T(Y) = \bigcup \{T(y) \mid y \in Y\}$$

Since the right-hand sides in both constraint system E and constraint system R denote monotonic functions, these systems of inequations have unique least solutions.

3 Analysis of Side-Effects

This section consists of three parts: Firstly, we describe how to embed the side-effect information of a procedure into any intraprocedural value analysis. Secondly, we present an interprocedural analysis which computes the modifying potential of each procedure. And, thirdly, we prove the correctness of our side-effect analysis.

3.1 Effect Integration

Our goal is to determine for every procedure q its *modifying* potential, i.e., the set of local memory cells whose contents are possibly modified during an invocation of q. For that, we consider the register values at a procedure call as the arguments of the procedure. The modifying potential $[\![q]\!]^{\natural}$ therefore, is represented by two components (X, M) where:

- $X \subseteq \mathbf{X}$ is a subset of registers whose values after the call are equal to their values before the call. This set should always contain $\mathbf{x}_1$.
- $M : \mathbf{X} \to 2^{\mathbb{Z}}$ is a mapping which for each register $\mathbf{x}_i$ provides a (super)set of all $\mathbf{x}_i$-relative write accesses to the local memory. In a concrete implementation, the analysis may compute with particular sets of offsets only, such as intervals [20,6] or strided intervals [1].

Given the modifying potential $[\![q]\!]^\natural$ of all procedures q in the program, a value analysis can be constructed which determines for every program point, for all registers and local memory locations a superset of their respective values. For that, we are interested in two kinds of values:

- *absolute values*, i.e., potential addresses in the global memory. These are represented as $(\mathbf{x}_0, z)$. Register $\mathbf{x}_0$ is used for accessing the segment of global memory. In our framework, we assume $\mathbf{x}_0$ to be hard-wired to the value $(G, 0)$.
- *stack pointer offsets*, i.e., local memory locations which are addressed relative to $\mathbf{x}_1$. These are represented as $(\mathbf{x}_1, z)$.

Thus, we consider the value domain $\mathbb{V} = 2^{\{\mathbf{x}_1,\mathbf{x}_0\}\times\mathbb{Z}}$ where the greatest element $\top$ is given by the set $\{\mathbf{x}_1, \mathbf{x}_0\} \times \mathbb{Z}$.

Let $\mathbf{S}^\sharp{}_q := [0, c_q]$ denote the set of all procedure-local memory cells of procedure q where the constant c_q is provided by the initial control flow edge of procedure q.

An abstract program state w.r.t. procedure q is then given by the triple $\langle\rho^\sharp, c_q, \lambda^\sharp\rangle$, with:

- $\rho^\sharp : \mathbf{X} \to \mathbb{V}$ which assigns the value $\{(\mathbf{x}_1, 0)\}$ to the stack pointer $\mathbf{x}_1$, and to each register $\mathbf{x}_i \in \mathbf{X}$ different from $\mathbf{x}_1$ a set of its possible values.
- c_q denotes the size of the stack frame of procedure q. This constant is provided by the single edge starting at the entry point of q.
- $\lambda^\sharp : \mathbf{S}^\sharp{}_q \to \mathbb{V}$ which assigns every local memory location from the stack frame of q a set of its possible values.

Thus, our analysis determines for every program point u of a procedure q a set of program states $\Gamma^\sharp$ of the form $\langle\rho^\sharp, c_q, \lambda^\sharp\rangle$. Again, $R^\sharp(u)$ is defined as the least solution of the following constraint system:

$$\begin{array}{lll}
[\mathrm{R}^\sharp 0]\ R^\sharp(s_q) \sqsupseteq \Gamma_0^\sharp & & s_q \text{ start point of procedure } q \\
[\mathrm{R}^\sharp 1]\ R^\sharp(v) \ \sqsupseteq [\![s]\!]^\sharp(R^\sharp(u)) & & (u, s, v) \text{ with } s \in \mathsf{stm} \cup \mathsf{guards} \\
[\mathrm{R}^\sharp 2]\ R^\sharp(v) \ \sqsupseteq [\![p]\!]^\natural \ @^\sharp \ (R^\sharp(u)) & & (u, p(), v)
\end{array}$$

where $\Gamma_0^\sharp$ sets $\mathbf{x}_1$ to the set $\{(\mathbf{x}_1, 0)\}$ and all other registers and local memory locations to the full set $\{\mathbf{x}_0, \mathbf{x}_1\} \times \mathbb{Z}$ of values. Thus:

$$\Gamma_0^\sharp = \{\gamma^\sharp : \langle \mathbf{X} \to \top, 0, \bot\rangle \mid \gamma^\sharp(\mathbf{x}_1) = \{(\mathbf{x}_1, 0)\}\}$$

where $\bot$ denotes the empty mapping. The transformers $[\![s]\!]^\sharp$ are the abstract effects of edge labels, $[\![p]\!]^\natural$ represents the modifying potential of procedure p, and $@^\sharp$ is the application of a modifying potential to a given assignment of registers and local addresses to sets of values. For registers $\mathbf{x}_i$, we have:

$$((X, M)\ @^\sharp\ \langle\rho^\sharp, c_q, \lambda^\sharp\rangle)(\mathbf{x}_i) = \begin{cases} \top & \text{if } \mathbf{x}_i \notin X \\ \rho^\sharp(\mathbf{x}_i) & \text{if } \mathbf{x}_i \in X \end{cases}$$

For a local offset a,

$$((X, M)\ @^\sharp\ \langle\rho^\sharp, c_q, \lambda^\sharp\rangle)(a) = \top$$

if there exists some $\mathbf{x}_i \in \mathbf{X}, d \in M(\mathbf{x}_i), (\mathbf{x}_1, b) \in \rho^\sharp(\mathbf{x}_i)$ such that $a = b + d$. Otherwise,

$$((X, M) \mathbin{@^\sharp} \langle \rho^\sharp, c_q, \lambda^\sharp \rangle)(a) = \lambda^\sharp(a)$$

i.e., remains unchanged. In case that for $\mathbf{x}_i \in \mathbf{X}, d \in M(\mathbf{x}_i)$ and $(\mathbf{x}_1, b) \in \rho^\sharp(\mathbf{x}_i)$, $b < d$, i.e., the offset $b + d$ is *negative*, a potential access to a local memory location outside the stack is detected. In this case, we again flag a warning and abort the analysis.

The abstract transformers $[\![s]\!]^\sharp$ are identical to the abstract transformers which we use for an auxiliary intraprocedural value analysis when computing the modifying potential of procedures.

Example 2. Recall the program from Example 1 and its corresponding control flow representation in Figure 1. According to our effect description the modifying potential of procedure f is given by $[\![f]\!]^\natural = (\{\mathbf{x}_1\}, \{(\mathbf{x}_3, 0), (\mathbf{x}_3, 4), (\mathbf{x}_3, 8), (\mathbf{x}_3, 12)\})$ at program point 23. At program point 6, directly before the procedure call, we have the following register assignment: $(\mathbf{x}_1, 8) \mapsto \{(\mathbf{x}_0, 13)\}; \mathbf{x}_2 \mapsto \{(\mathbf{x}_1, 12)\}; \mathbf{x}_3 \mapsto \{(\mathbf{x}_1, 12)\}$. Embedding the side-effect information of f into the intraprocedural analysis of procedure main, then directly after the procedure call $f()$ the value of the stack location $(\mathbf{x}_1, 8)$ remains unchanged. Thus we arrive at the following register assignment for program point 7: $(\mathbf{x}_1, 8) \mapsto \{(\mathbf{x}_0, 13)\}$. □

Next we describe how to compute the modifying potential of a procedure.

3.2 Effect Computation

Assume that we are given a mapping μ which assigns to each procedure g an approximation of its modifying potential, i.e., $\mu(g) = (X_g, M_g)$ where $X_g \subseteq \mathbf{X}$ and $M_g : \mathbf{X} \to 2^{\mathbb{Z}}$. Relative to μ, we perform for a given procedure g, an intraprocedural *value* analysis which determines for every program point u of g and all registers $\mathbf{x}_i \in \mathbf{X}$ as well as all g-local memory cells $a \in [0, c_g]$, sets $\rho^\natural(u)(\mathbf{x}_i), \lambda^\natural(u)(a) \subseteq 2^{(\mathbf{X} \cup \{\mathbf{x}_0\}) \times \mathbb{Z}}$ of values relative to the values of registers at procedure entry. Again c_g denotes the stack frame size of procedure g. Relative to the mappings $\langle \rho^\natural(u), \lambda^\natural(u) \rangle$, u a program point of g, the modifying potential of g is given by $\mathsf{eff}_g(\mu) = (X', M')$ where

$$\begin{array}{rl} X' &= \{\mathbf{x}_i \in \mathbf{X} \mid \rho^\natural(r_g)(\mathbf{x}_i) = \{(\mathbf{x}_i, 0)\}\} \\ M' &= (\mathbf{X} \times 2^{\mathbb{Z}}) \cap \big(\bigcup\{[\![t]\!]^\natural(\rho^\natural \mid (u, M[t] := \mathbf{x}_j, \_) \in E_g)\} \cup \\ & \qquad \{(\mathbf{x}_k, z + z') \mid (u, f(), \_) \in E_g, (\mathbf{x}_k, z) \in \rho^\natural(u)(\mathbf{x}_{k'}), (\mathbf{x}_{k'}, z') \in M_f\}\big) \end{array}$$

Note that (X', M') monotonicly depends on μ, only if the mappings $\rho^\natural(u)$ as well as $\lambda^\natural(u)$ monotonicly depend on μ. Thus, given such a monotonic value analysis of the functions $\rho^\natural, \lambda^\natural$, the modifying potential can be determined as the least (or some) solution of the constraint system:

$$[\![g]\!]^\natural \sqsupseteq \mathsf{eff}_g((\emptyset, \emptyset)), \qquad \text{for all procedures } g$$

It remains to construct the constraint system whose least solution characterises the mappings $\rho^\natural(u) : \mathbf{X} \to \bar{\mathbb{V}}$ and $\lambda^\natural : [0, c_g] \to \bar{\mathbb{V}}$ w.r.t. procedure g with stack frame size c_g.

Now, $\bar{\mathbb{V}}$ is the complete lattice $2^{(\mathbf{X} \cup \{\mathbf{x}_0\}) \times \mathbb{Z}}$. Note that the greatest element $\top$ of $\bar{\mathbb{V}}$ is given by $\top = (\mathbf{X} \cup \{\mathbf{x}_0\}) \times \mathbb{Z}$.

The abstract arithmetic operations are obtained by first considering single elements $(\mathbf{x}_i, c)$. We define:

$$\begin{aligned} (\mathbf{x}_i, c_1) \odot (\mathbf{x}_0, c_2) &= \{(\mathbf{x}_i, c_1 \odot c_2)\} \\ (\mathbf{x}_0, c_2) \odot (\mathbf{x}_i, c_1) &= \{(\mathbf{x}_i, c_1 \odot c_2)\} \\ (\mathbf{x}_i, c_1) \odot (\mathbf{x}_j, c_2) &= \top \end{aligned}$$

for $\odot \in \{+, \cdot\}$, while for subtraction we define:

$$\begin{aligned} (\mathbf{x}_i, c_1) - (\mathbf{x}_i, c_2) &= \{(\mathbf{x}_0, c_1 - c_2)\} \\ (\mathbf{x}_i, c_1) - (\mathbf{x}_0, c_2) &= \{(\mathbf{x}_i, c_1 - c_2)\} \\ (\mathbf{x}_i, c_1) - (\mathbf{x}_j, c_2) &= \top \end{aligned}$$

for $i \neq j$. These definitions then are lifted to abstract operators $\odot^\natural$ on sets of such elements, i.e., to $\bar{\mathbb{V}}$ by:

$$S_1 \odot^\natural S_2 = \bigcup \{s_1 \odot s_2 \mid s_1 \in S_1, s_2 \in S_2\}$$

The definition of these abstract operators gives rise to an abstract evaluation $[\![t]\!]^\natural$ of expressions t w.r.t. to a given register assignment $\rho^\natural$.

$$[\![t]\!]^\natural(\rho^\natural) = \begin{cases} [\![t_1]\!]^\natural(\rho^\natural) \odot^\natural [\![t_2]\!]^\natural(\rho^\natural) & \text{if } t = t_1 \odot t_2 \\ \rho^\natural(\mathbf{x}_i) & \text{if } t = \mathbf{x}_i \\ \{(\mathbf{x}_0, c)\} & \text{if } t = c \end{cases}$$

The mappings $\rho^\natural(u), \lambda^\natural(u)$ for a procedure g then can be characterised by the least solution of the following constraint system:

$$\begin{array}{llll} [\mathrm{R}^\natural_\mu 0] & R^\natural_\mu(s_g) & \sqsupseteq \langle \{\mathbf{x}_i \mapsto \{(\mathbf{x}_i, 0)\} \mid \mathbf{x}_i \in \mathbf{X} \cup \{\mathbf{x}_0\}\}, 0, \bot \rangle & \\ [\mathrm{R}^\natural_\mu 1] & R^\natural_\mu(v) & \sqsupseteq [\![s]\!]^\natural(R^\natural_\mu(u)) & (u, s, v) \text{ with } s \in \textsf{stm} \cup \textsf{guards} \\ [\mathrm{R}^\natural 2] & R^\natural_\mu(v) & \sqsupseteq (R^\natural_\mu(r_f))@^\natural(R^\natural_\mu(u)) & (u, f(), v) \text{ a call edge} \end{array}$$

At procedure start, all registers $\mathbf{x}_i \in \mathbf{X}$ are mapped to their symbolic values $\{(\mathbf{x}_i, 0)\}$ (constraint $[\mathrm{R}^\natural 0]$).

The effect of the instruction decrementing the stack pointer is that a new stack frame $\mathbf{S}^\sharp{}_g$, i.e., the local memory locations $\mathbf{S}^\sharp{}_g$, for procedure g is allocated. Thus, we have:

$$[\![\mathbf{x}_1 := \mathbf{x}_1 - c]\!]^\natural(\langle \rho^\natural, c', \lambda^\natural \rangle) \;=\; \langle \rho^\natural, c, \{a \mapsto \top \mid a \in [0, c]\} \rangle$$

The effect of the instruction incrementing the stack pointer is that the stack frame $\mathbf{S}^\sharp{}_g$ for procedure g is deallocated. Thus, the set of variables is restricted to registers only:

$$[\![\mathbf{x}_1 := \mathbf{x}_1 + c]\!]^\natural(\langle \rho^\natural, c, \lambda^\natural \rangle) \;=\; \langle \rho^\natural, 0, \bot \rangle$$

The second constraint $[\mathrm{R}^\natural 1]$ handles assignments to other registers, memory accesses and guards. For assignments, we have:

$$\begin{aligned} [\![\mathbf{x}_i := ?]\!]^\natural(\langle \rho^\natural, c, \lambda^\natural \rangle) \;&=\; \langle \rho^\natural \oplus \{\mathbf{x}_i \mapsto \top\}, c, \lambda^\natural \rangle \\ [\![\mathbf{x}_i := t]\!]^\natural(\langle \rho^\natural, c, \lambda^\natural \rangle) \;&=\; \langle \rho^\natural \oplus \{\mathbf{x}_i \mapsto [\![t]\!]^\natural(\rho^\natural)\}, c, \lambda^\natural \rangle \end{aligned}$$

with $i \neq 1$. The effect of a *memory read access* instruction in procedure g on a state $\langle \rho^\natural, c, \lambda^\natural \rangle$ is given by:

$$[\![\mathbf{x}_i := M[t]]\!]^\natural(\langle \rho^\natural, c, \lambda^\natural \rangle) =$$

$$\begin{cases} \langle \rho^\natural \oplus \{\mathbf{x}_i \mapsto \top_G\}, c, \lambda^\natural \rangle & \text{if } [\![t]\!]^\natural(\rho^\natural) \subseteq \{\mathbf{x}_0\} \times \mathbb{Z} \\ \langle \rho^\natural \oplus \{\mathbf{x}_i \mapsto \bigcup\{\lambda^\natural(c') \mid (\mathbf{x}_1, c') \in [\![t]\!]^\natural(\rho^\natural)\}\}, c, \lambda^\natural \rangle & \text{if } [\![t]\!]^\natural(\rho^\natural) \subseteq \{\mathbf{x}_1\} \times [0, c] \\ \langle \rho^\natural \oplus \{\mathbf{x}_i \mapsto \top\}, c, \lambda^\natural \rangle & \text{otherwise} \end{cases}$$

with $i \neq 1$. The *global top* $\top_G$ describes the set of all possible global addresses: $\top_G = \mathbf{x}_0 \times \mathbb{Z}$.

Since we do not track the values of global variables, in case of a memory access to a global memory location, variable $\mathbf{x}_i$ may be assigned every possible global value. If the evaluation of a memory access expression yields that a local variable $(\mathbf{x}_1, c)$ is addressed which belongs to the stack frame of the current procedure, its value is assigned to register $\mathbf{x}_i$. For all other cases the value of $\mathbf{x}_i$ is overapproximated by the top element.

For a *memory write* instruction the abstract effect function is defined by:

$$[\![M[t] := \mathbf{x}_j]\!]^\natural(\langle \rho^\natural, c, \lambda^\natural \rangle) =$$

$$\begin{cases} \langle \rho^\natural, c, \lambda^\natural \oplus \{c' \mapsto \rho^\natural(\mathbf{x}_j)\}\rangle & \text{if } \{(\mathbf{x}_1, c')\} = [\![t]\!]^\natural(\rho^\natural) \wedge c' \in [0, c] \\ \langle \rho^\natural, c, \lambda^\natural \oplus \{c' \mapsto (\lambda^\natural(c') \cup \rho^\natural(\mathbf{x}_j)) \mid (\mathbf{x}_1, c') \in [\![t]\!]^\natural(\rho^\natural), c' \in [0, c]\} & \text{otherwise} \end{cases}$$

with $j \neq 1$. If the accessed memory location denotes a single local variable of the current stack frame the value of variable $\mathbf{x}_j$ is assigned to the corresponding local memory location. If the evaluation of a memory access expression yields a set of possibly accessed local memory locations, all their values are extended by the value of variable $\mathbf{x}_j$. In all the other cases, none of the local memory locations may receive new values. If an element $(\mathbf{x}_1, c')$ is found in $[\![t]\!]^\natural(\rho^\natural)$ with $c' \notin [0, c]$, we issue a warning and abort the analysis.

Guards are used for restricting the sets of possible values of registers. Sets of possible values, however, can only be compared if they refer to the same base register. First let us consider the guard $\mathbf{x}_i \bowtie c$. If $\rho^\natural(\mathbf{x}_i) = \{\mathbf{x}_0\} \times S$ and $c \in \mathbb{Z}$, let $S' = \{s \in S \mid s \bowtie c\}$. Then we set:

$$[\![\mathbf{x}_i \bowtie c]\!]^\natural(\langle \rho^\natural, c', \lambda^\natural \rangle) = \langle \rho^\natural \oplus \{\mathbf{x}_i \mapsto \{\mathbf{x}_0\} \times S'\}, c', \lambda^\natural \rangle$$

Likewise, for a guard $\mathbf{x}_i \bowtie \mathbf{x}_j$, if $\rho^\natural(\mathbf{x}_i) = \{\mathbf{x}_k\} \times S_1$ and $\rho^\natural(\mathbf{x}_j) = \{\mathbf{x}_k\} \times S_2$, then we set

$$\begin{aligned} S_1' &= \{s \in S_1 \mid \exists s_2 \in S_2 : s \bowtie s_2\} \\ S_2' &= \{s \in S_2 \mid \exists s_1 \in S_1 : s_1 \bowtie s\} \end{aligned}$$

and define

$$[\![\mathbf{x}_i \bowtie \mathbf{x}_j]\!]^\natural(\langle \rho^\natural, c', \lambda^\natural \rangle) = \langle \rho^\natural \oplus \{\mathbf{x}_i \mapsto \{\mathbf{x}_k\} \times S_1', \mathbf{x}_j \mapsto \{\mathbf{x}_k\} \times S_2'\}, c', \lambda^\sharp \rangle$$

In all other cases, guards have no effect on the register assignment.

3.3 Correctness

The correctness proof is based on a description relation $\Delta \subseteq \Gamma \times \Gamma^\sharp$ between concrete and abstract states. A description relation is a relation with the following property: $s\,\Delta\,s_1^\sharp \;\wedge\; s_1^\sharp \sqsubseteq s_2^\sharp \Rightarrow s\,\Delta\,s_2^\sharp$ for $s \in \Gamma$ and $s_1^\sharp, s_2^\sharp \in s^\sharp$.
Here, the concrete state $\langle \rho, f, \lambda \rangle$ is described by the abstract state $\langle \rho^\sharp, c, \lambda^\sharp \rangle \in \Gamma^\sharp$ if

- $f = (x_r, x_{r-1}, \ldots, x_0)$ with $x_{r-1} - x_r = c$;
- $\rho(\mathbf{x}_i) \in \gamma_{x_r}(\rho^\sharp(\mathbf{x}_i))$ for all $\mathbf{x}_i$;
- $\lambda(L, a) \in \gamma_{x_r}(\lambda^\sharp(a - x_r))$.

Here, the concretisation γ_x replaces the tagging register $\mathbf{x}_0$ with G, while symbolic local addresses $(\mathbf{x}_1, b)$ are translated into $(L, x - b)$.

Additionally, we require a description relation between the transformations induced by same-level concrete computations and abstract states from $R_\mu^\natural$. Assume that $T : \Gamma \to 2^\Gamma$ is a transformation which preserves frame structures, i.e., $f = f'$ for all $\langle \rho', f', \lambda' \rangle \in T(\langle \rho, f, \lambda \rangle)$. Then T is described by $\langle \rho^\natural, c, \lambda^\natural, X, M \rangle$ if for all $\langle \rho', f', \lambda' \rangle \in T(\langle \rho, f, \lambda \rangle)$,

- $f = (x_r, x_{r-1}, \ldots, x_0)$ where $x_{r-1} - x_r = c$;
- $\rho'(\mathbf{x}_i) \in \{\rho(\mathbf{x}_j) + (G, a) \mid (\mathbf{x}_j, a) \in \rho^\natural(\mathbf{x}_i)\}$ for all $\mathbf{x}_i$;
- $\lambda'(L, a) \in \{\rho(\mathbf{x}_j) + (G, a) \mid (\mathbf{x}_j, a) \in \lambda^\natural(a - x_r)\}$ for all $a \in [x_r, x_{r-1})$;
- $\rho(\mathbf{x}_i) = \rho'(\mathbf{x}_i)$ for all $\mathbf{x}_i \in X$;
- If $\lambda(L, b) \neq \lambda'(L, b)$ for $b \geq x_{r-1}$, then $(L, b) = \rho(\mathbf{x}_i) + (G, a)$ for some $(\mathbf{x}_i, a) \in M$.

Here, we have assumed that $\rho(\mathbf{x}_0)$ always returns $(G, 0)$. The description relation for transformers is preserved by function composition, execution of individual statements as well as least upper bounds. Therefore, we obtain by induction on the fixpoint iterates:

Theorem 1. Correctness

1. *Assume that $E, R_\mu^\natural$ denote the least solutions of the constraint systems for the concrete effects of procedures and their side effects, respectively. Then for every program point u, the transformer $E(u)$ is frame preserving, and $E(u)\ \Delta\ R_\mu^\natural(u)$.*
2. *Let $R, R^\sharp$ denote the least solutions of the constraint systems for the collecting semantics and abstract reachability, respectively. Then for every program point u, $R(u)\ \Delta\ R^\sharp(u)$.*

Local Addresses Escaping to the Heap

Although the description of the collecting semantics excludes those programs, where e.g. local addresses may be written into global memory, our abstract approach is able to detect such situations. Therefore, the modifying potential (X, M) of a procedure can be enriched by an additional component η, where

$$\eta : (\mathbf{X} \cup \{\mathbf{x}_0\}) \to 2^{\mathbf{X}}$$

η is a mapping which assigns to each register $\mathbf{x}_i$ the subset of registers $\mathbf{x}_j$ such that $\mathbf{x}_j$-relative addresses may escape to the global memory if $\mathbf{x}_i$ is a global address. Provided an abstract value analysis which maps registers to some abstract values ($\rho^\natural$) and local memory locations to some abstract values ($\lambda^\natural$), the effect computation for η is given by:

$$\begin{aligned}\eta'(\mathbf{x}_i) = \{&\mathbf{x}_k \in \mathbf{X} \mid \exists(u, M[t] := \mathbf{x}_j, \_) \in E_g, \exists z, z' \in \mathbb{Z}.(\mathbf{x}_k, z) \in V_\mu(u)(\mathbf{x}_j)\\ &\wedge(\mathbf{x}_i, z') \in \llbracket t \rrbracket^\natural(V_\mu(u))\}\cup\\ \{&\mathbf{x}_j \in \mathbf{X} \mid (u, f(), \_) \in E_g, (\mathbf{x}_i, z) \in V_\mu(u)(\mathbf{x}_k), \mathbf{x}_{k'} \in \eta_f(\mathbf{x}_k),\\ &(\mathbf{x}_j, z') \in V_\mu(u)(\mathbf{x}_{k'})\}\end{aligned}$$

Now the modifying potential of a procedure g, i.e. the triple (X, M, η), can be determined as the least solution of the constraint system:

$$\llbracket g \rrbracket^\natural \sqsupseteq \mathsf{eff}_g((\emptyset, \bot, \emptyset)), \qquad \text{for all procedures } g$$

The component η serves as another soundness check. If a register $\mathbf{x}_i$ may hold a global address, i.e., $\rho^\natural(\mathbf{x}_i)$ contains an element $(\mathbf{x}_0, z)$, and $\mathbf{x}_j \in \eta(\mathbf{x}_i)$ is found such that $\rho^\natural(\mathbf{x}_j)$ contains a stack address, i.e., an element $(\mathbf{x}_1, z')$, then some reference to the stack may have escaped to the global memory. In this case, we flag a warning and again abort the analysis.

4 Experimental Results

We have implemented the side-effect analysis in our assembly analyser VoTUM [30]. Our benchmark suite consists of statically linked PowerPC assemblies of publicly available programs such as the cryptography library **openSSL**, and the HTTP server **thttpd**. Moreover, we experimented with the four larger programs basename, vdir, chmod, chgrp from the Unix GNU Coreutils package, and also ran the prototype on **gzip** from SPECint. The vehicle control **control**[29] is generated from SCADE and thus is very restrictive in the use of pointers and does not use dynamic memory [9]. The binaries of these programs have sizes between 0.6 and 3.8 MB and have been compiled with gcc version 4.4.3 at optimisation levels $O0$ and $O2$ and are statically linked with glibc. We conducted our experiments on a 2.2 GHz quad-core machine equipped with 16 GB of physical memory where the algorithm occupies just a single core. We have implemented our side-effect analysis with the enhancement of Section 3.3. Additionally, we track the values of global memory locations with static addresses. We have evaluated the significance of our side-effect analysis by means of:

- The percentage of side-effect free procedures, i.e. those procedures that have no effect to the stack frame of other procedures, in the binary (Column **SE_free**).
- The absolute number of procedures where locals may escape to the heap (Column **Esc**).
- The percentage of locals that have to be invalidated after a procedure call and in parentheses the absolute number of call sites with side-effects (Column **Inv**).
- The absolute number of procedures where a safe handling of the link register and the stack pointer could not be verified (Columns **LR** and **SP**).

- The percentage of procedures where our analysis infers unbounded side-effects (Column **Top**). In contrast to our theoretical conception from Section 3 we follow a less restrictive approach in our implementation: in case of an unbounded side-effect our analyser reports a warning and proceeds by setting all the values of the registers of the caller to unknown values except for the stack pointer and the link register.

Evaluating our experimental results, we have: Column **SE_free** reveals that $20\% - 50\%$ of all the procedures in the binary *have side-effects*. However, side-effect information concerns only the local variables of up to 2% of all call sites (Column **Inv**). Embedding the side-effects into the caller invalidates a third of the caller‘s locals in average. Column **Esc** illustrates that stack addresses rarely escape to global memory. This is the case for approximately 0.5% of all procedures in our benchmark suite. For at most 2% of all the procedures our analysis could not verify the integrity of the organisational stack cells, cf. Columns **LR** and **SP**. These procedures are either related to error handling or require relational domains to be analysed precisely. For instance when the bound for iterating over an array is provided as the parameter of a procedure, domains like polyhedra [7] or symbolic intervals [26] have to be used. For at most 9% of all procedures our analysis inferred an unbounded side-effect (Column **Top**). In order to achieve more precision in these cases, we have to instantiate our side-effect analysis with more sophisticated domains [26,7] and better widening strategies (for sets of values). For our benchmark programs, the analyser consumes between 3 and 15 GB memory and analysis time is between 20 minutes up to 1.5 hours. In order to rate the light-weightness of our approach, we want to mention that the control flow reconstruction analysis of that large binaries has approximately the same running time.

Program	Size	Instr	Procs	Calls	SE_free	Esc	Inv	LR	SP	Top	M	T
openSSL_O0	3.8MB	769511	6710	38188	78%	27	22%(549)	152	21	4%	15168	2298
thttpd_O0	0.8MB	196493	1200	7947	48%	2	27% (115)	28	17	9%	6157	1189
basename_O0	0.7MB	168232	916	5303	53%	5	33% (89)	8	5	8%	3387	1417
chgrp_O0	0.8MB	197260	1119	6338	51%	5	22% (192)	11	5	8%	4264	2052
chmod_O0	0.7MB	179415	1027	5767	52%	1	38% (96)	24	12	7%	3609	1739
vdir_O0	0.9MB	221665	1307	6988	56%	3	25% (165)	45	15	7%	5329	1060
gzip_O0	0.8MB	166213	1078	5954	53%	2	30%(112)	30	10	7%	3924	1858
control_O0	0.6MB	162530	819	4948	49%	5	31%(112)	9	8	9%	3284	2197
openSSL_O2	2.9MB	613882	6234	43407	72%	6	30%(486)	464	20	4%	15396	5594
thttpd_O2	0.8MB	189034	1150	5706	48%	1	19%(348)	44	13	9%	5149	1906
basename_O2	0.7MB	139271	907	5386	51%	1	30%(61)	23	6	8%	3326	1555
chgrp_O2	0.7MB	164420	1083	6522	49%	5	23%(199)	7	7	7%	4086	1234
chmod_O2	0.7MB	148702	989	6005	50%	5	29%(123)	8	7	7%	3749	1629
vdir_O2	0.8MB	177022	1200	7106	51%	2	35%(80)	40	7	7%	5152	1304
gzip_O2	0.7MB	162380	1028	6323	52%	1	33%(113)	28	6	8%	3960	1841
control_O2	0.6MB	161184	819	4989	49%	5	33%(81)	8	5	9%	3286	2380

Size: the binary file size; **Instr**: the number of assembler instructions;
Procs: the number of procedures; **Calls**: the number of procedure calls;
T: the absolute running time in seconds; **M**: the peak memory consumption in MB.

Summarising, for most of the procedures our side-effect analysis provides tight bounds for parameter relative write accesses even with the simplistic instantiation of our value analysis with sets of values. Since approximately 50% of all procedures have side-effects, for a sound and precise analysis it is mandatory to consider side-effect information.

5 Conclusion

We have presented a reasonably fast interprocedural analysis of assembly code for inferring the side-effects of procedure calls onto the runtime stack. Such an analysis contributes in refining arbitrary analyses by identifying possible modifications in the stack frame of the caller through a procedure call. Our analysis allows to verify in how far basic assumptions for the analysis of the assembly code are met. Still, precision could be significantly enhanced, if the analysis tracked relational invariants for values. Our approach can be extended to a *purity analysis* which not only tracks modifications to the stack but also to the heap. In order to deal with assembly code generated from, e.g. object-oriented programs, it also seems inevitable to combine our methods with simple forms of heap analysis such as [2].

References

1. Balakrishnan, G., Reps, T.: Recovery of Variables and Heap Structure in x86 Executables. Technical report, University of Wisconsin, Madison (2005)
2. Balakrishnan, G., Reps, T.: Recency-abstraction for heap-allocated storage. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 221–239. Springer, Heidelberg (2006)
3. Banning, J.P.: An efficient way to find the side effects of procedure calls and the aliases of variables. In: POPL 1979: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 29–41. ACM, New York (1979)
4. Choi, J.-D., Burke, M., Carini, P.: Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In: POPL 1993: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 232–245. ACM, New York (1993)
5. Cooper, K.D., Kennedy, K.: Interprocedural side-effect analysis in linear time. In: PLDI 1988: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, pp. 57–66. ACM, New York (1988)
6. Cousot, P., Cousot, R.: Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In: Bruynooghe, M., Wirsing, M. (eds.) PLILP 1992. LNCS, vol. 631, pp. 269–295. Springer, Heidelberg (1992)
7. Cousot, P., Halbwachs, N.: Automatic Discovery of Linear Restraints among Variables of a Program. In: 5th Ann. ACM Symposium on Principles of Programming Languages (POPL), pp. 84–97 (1978)
8. Debray, S., Muth, R., Weippert, M.: Alias analysis of executable code. In: POPL 1998: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 12–24. ACM, New York (1998)
9. Dormoy, F.-X., Technologies, E.: SCADE 6 A Model Based Solution For Safety Critical Software Development (2008), http://www.esterel-technologies.com/technology/WhitePapers/

10. Dullien, T., Porst, S.: REIL: A platform-independent intermediate representation of disassembled code for static code analysis (2009), http://www.zynamics.com/downloads/csw09.pdf
11. Emami, M., Ghiya, R., Hendren, L.J.: Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, PLDI 1994, pp. 242–256. ACM, New York (1994)
12. Flexeder, A., Mihaila, B., Petter, M., Seidl, H.: Interprocedural control flow reconstruction. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 188–203. Springer, Heidelberg (2010)
13. Flexeder, A., Petter, M., Seidl, H.: Analysis of executables for WCET concerns. Technical Report, Institutfür Informatik (2008), http://www2.in.tum.de/flexeder/report38.pdf
14. Frey, B.: PowerPC Architecture Book, Version 2.02 (November 2005), http://www.ibm.com/developerworks/systems/library/es-archguide-v2.html
15. Guo, B., Bridges, M.J., Triantafyllis, S., Ottoni, G., Raman, E., August, D.I.: Practical and Accurate Low-Level Pointer Analysis. In: CGO 2005: Proceedings of the International Symposium on Code Generation and Optimization, pp. 291–302. IEEE Computer Society, Washington, DC, USA (2005)
16. Kinder, J., Veith, H.: Jakstab: A static analysis platform for binaries. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 423–427. Springer, Heidelberg (2008)
17. Kinder, J., Zuleger, F., Veith, H.: An abstract interpretation-based framework for control flow reconstruction from binaries. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 214–228. Springer, Heidelberg (2009)
18. Landi, W., Ryder, B.G., Zhang, S.: Interprocedural Modification Side Effect Analysis With Pointer Aliasing. In: Proceedings of the SIGPLAN 1993 Conference on Programming Language Design and Implementation, pp. 56–67 (1993)
19. Linn, C., Debray, S., Andrews, G., Schwarz, B.: Stack Analysis of x86 Executables (2004), http://www.cs.arizona.edu/ debray/Publications/stack-analysis.pdf
20. Moore, R.E., Bierbaum, F.: Methods and Applications of Interval Analysis (SIAM Studies in Applied and Numerical Mathematics) (Siam Studies in Applied Mathematics, 2). Soc. for Industrial & Applied Math., Philadelphia (1979)
21. Müller-Olm, M., Seidl, H.: Precise Interprocedural Analysis through Linear Algebra. In: 31st ACM Symp. on Principles of Programming Languages (POPL), pp. 330–341 (2004)
22. Müller-Olm, M., Seidl, H.: Upper adjoints for fast inter-procedural variable equalities. In: Gairing, M. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 178–192. Springer, Heidelberg (2008)
23. Reps, T., Balakrishnan, G.: Improved memory-access analysis for x86 executables. In: Hendren, L. (ed.) CC 2008. LNCS, vol. 4959, pp. 16–35. Springer, Heidelberg (2008)
24. Reps, T., Balakrishnan, G., Lim, J.: Intermediate-representation recovery from low-level code. In: PEPM 2006: Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, pp. 100–111. ACM, New York (2006)
25. Sălcianu, A., Rinard, M.C.: Purity and side effect analysis for java programs. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 199–215. Springer, Heidelberg (2005)
26. Sankaranarayanan, S., Ivancic, F., Gupta, A.: Program analysis using symbolic ranges. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 366–383. Springer, Heidelberg (2007)
27. Sharir, M., Pnueli, A.: Two Approaches to Interprocedural Data Flow Analysis. In: Program Flow Analysis: Theory and Application, pp. 189–234 (1981)

28. Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: BitBlaze: A new approach to computer security via binary analysis. In: Sekar, R., Pujari, A.K. (eds.) ICISS 2008. LNCS, vol. 5352, pp. 1–25. Springer, Heidelberg (2008)
29. Sicherheitsgarantien Unter REALzeitanforderungen (2010), http://www.sureal-projekt.org/
30. VoTUM (2010), http://www2.in.tum.de/votum
31. Wilson, R.P., Lam, M.S.: Efficient context-sensitive pointer analysis for C programs. In: PLDI 1995: Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation, pp. 1–12. ACM, New York (1995)

Directed Symbolic Execution

Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks

Computer Science Department, University of Maryland, College Park
{kkma,khooyp,jfoster,mwh}@cs.umd.edu

Abstract. In this paper, we study the problem of automatically finding program executions that reach a particular target line. This problem arises in many debugging scenarios; for example, a developer may want to confirm that a bug reported by a static analysis tool on a particular line is a true positive. We propose two new *directed* symbolic execution strategies that aim to solve this problem: *shortest-distance symbolic execution (SDSE)* uses a distance metric in an interprocedural control flow graph to guide symbolic execution toward a particular target; and *call-chain-backward symbolic execution (CCBSE)* iteratively runs forward symbolic execution, starting in the function containing the target line, and then jumping backward up the call chain until it finds a feasible path from the start of the program. We also propose a hybrid strategy, Mix-CCBSE, which alternates CCBSE with another (forward) search strategy. We compare these three with several existing strategies from the literature on a suite of six GNU Coreutils programs. We find that SDSE performs extremely well in many cases but may fail badly. CCBSE also performs quite well, but imposes additional overhead that sometimes makes it slower than SDSE. Considering all our benchmarks together, Mix-CCBSE performed best on average, combining to good effect the features of its constituent components.

1 Introduction

In this paper, we study the *line reachability problem*: given a target line in the program, can we find a realizable path to that line? Since program lines can be guarded by conditionals that check arbitrary properties of the current program state, this problem is equivalent to the very general problem of finding a path that causes the program to enter a particular state [12]. The line reachability problem arises naturally in several scenarios. For example, users of static-analysis-based bug finding tools need to *triage* the tools' bug reports—determine whether they correspond to actual errors—and this task often involves checking line reachability. As another example, a developer might receive a report of an error at some particular line (e.g., an assertion failure that resulted in an error message at that line) without an accompanying test case. To reproduce the error, the developer needs to find a realizable path to the appropriate line. Finally, when trying to understand an unfamiliar code base, it is often useful to discover under what circumstances particular lines of code are executed.

E. Yahav (Ed.): SAS 2011, LNCS 6887, pp. 95–111, 2011.

Symbolic execution is an attractive approach to solving line reachability: by design, symbolic executors are *complete*, meaning any path they find is realizable. Symbolic executors work by running the program, computing over both concrete values and expressions that include *symbolic values*, which are unknowns that range over various sets of values, e.g., integers, strings, etc. [17,2,15,29]. When a symbolic executor encounters a conditional whose guard depends on a symbolic value, it invokes a theorem prover (our implementation uses the SMT solver STP [10]) to determine which branches are feasible. If both are, the symbolic execution conceptually forks, exploring both branches.

However, symbolic executors cannot explore all program paths, and hence must make heuristic choices to prioritize path exploration. Our work focuses on finding paths that reach certain lines in particular, whereas most prior work has focused on finding paths to increase code coverage [11,5,4,24,3,34]. We are aware of one previously proposed approach, *execution synthesis* (ESD) [36], for using symbolic execution to solve the line reachability problem; we compare ESD to our work in Section 3.

We propose two new *directed* symbolic execution search strategies for line reachability. First, we propose *shortest-distance symbolic execution* (SDSE), which prioritizes the path with the shortest distance to the target line as computed over an interprocedural control-flow graph (ICFG). Variations of this heuristic can be found in existing symbolic executors—in fact, SDSE is inspired by the heuristic used in the coverage-based search strategy from KLEE [4]—but, as far as we are aware, the strategy we present has not been specifically described nor has it been applied to directed symbolic execution. In Section 2.2 we describe how distance can be computed context-sensitively using *PN* grammars [32,9,30].

Second, we propose *call-chain-backward symbolic execution (CCBSE)*, which starts at the target line and works backward until it finds a realizable path from the start of the program, using standard forward (interprocedural) symbolic execution as a subroutine. More specifically, suppose the target line ℓ is inside function f. CCBSE begins forward symbolic execution from the start of f, yielding a set of partial interprocedural paths $\overline{p}_f$ that start at f, possibly call other functions, and lead to ℓ; in a sense, these partial paths summarize selected behavior of f. Next, CCBSE runs forward symbolic execution from the start of each function g that calls f, searching for paths that end at calls to f. For each such path p, it attempts to continue down paths p' in $\overline{p}_f$ until reaching ℓ, adding all feasible extended paths $p + p'$ to $\overline{p}_g$. The process continues backward up the call chain until CCBSE finds a path from the start of the program to ℓ. Notice that by using partial paths to summarize function behavior, CCBSE can reuse the machinery of symbolic execution to concatenate paths together. This is technically far simpler than more standard approaches that use some formal language to explicitly summarize function behavior in terms of parameters, return value, global variables, and the heap (including pointers and aliasing).

The key insight motivating CCBSE is that the closer forward symbolic execution starts relative to the target line, the better the chance it finds paths to that line. If we are searching for a line that is only reachable on a few paths along which many branches are possible, then combinatorially there is a very small chance that a standard symbolic executor will make the right choices and find that line. By starting closer to the line we are searching for, CCBSE explores shorter paths with fewer branches, and so is more likely to reach that line.

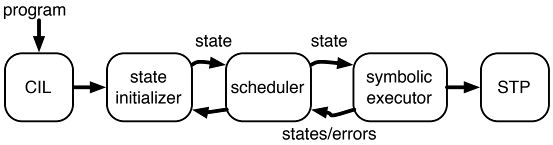
CCBSE imposes some additional overhead, and so it does not always perform as well as a forward execution strategy. Thus, we also introduce *mixed-strategy CCBSE (Mix-CCBSE)*, which combines CCBSE with another forward search. In Mix-CCBSE, we alternate CCBSE with some forward search strategy S. If S encounters a path p that was constructed in CCBSE, we try to follow p to see if we can reach the target line, in addition to continuing S normally. In this way, Mix-CCBSE can perform better than CCBSE and S run separately—compared to CCBSE, it can jump over many function calls from the program start to reach the paths being constructed; and compared to S, it can short-circuit the search once it encounters a path built up by CCBSE.

We implemented SDSE, CCBSE, and Mix-CCBSE in Otter, a C source code symbolic executor we previously developed [31]. We also extended Otter with two popular forward search strategies from KLEE [4] and SAGE [13], and for a baseline, we implemented a random path search that flips a coin at each branch. We evaluated the effectiveness of our directed search strategies on the line reachability problem, comparing against the existing search strategies. We ran each strategy on 6 different GNU Coreutils programs [6], looking in each program for one line that contains a previously identified fault. We also compared the strategies on synthetic examples intended to illustrate the strengths of SDSE and CCBSE. We found that SDSE performs extremely well on many programs, but it can fail completely under certain program patterns. CCBSE has performance comparable to standard search strategies but is often somewhat slower due to the overhead of checking path feasibility. Mix-CCBSE performs best of all across all benchmarks, particularly when using KLEE as its forward search strategy, since it exploits the best features of CCBSE and forward search. These results suggest that directed symbolic execution is a practical and effective approach to solving the line reachability problem.

2 Directed Symbolic Execution

In this section we present SDSE, CCBSE, and Mix-CCBSE. We will explain them in terms of their implementation in Otter, our symbolic execution framework, to make our explanations concrete (and to save space), but the ideas apply to any symbolic execution tool [17,11,4,16].

Figure 1 diagrams the architecture of Otter and gives pseudocode for its main scheduling loop. Otter uses CIL [27] to produce a control-flow graph from the input C program. Then it calls a *state initializer* to construct an initial symbolic execution *state*, which it stores in worklist, used by the scheduler. A state includes the stack, heap, program counter, and path taken to reach the current position. In traditional symbolic execution, which we call *forward* symbolic execution, the initial state begins execution at the start of main. The scheduler extracts a



```
scheduler()
  while (worklist nonempty)
    $s_0$ = pick(worklist)
    for $s \in$ step($s_0$) do
      if ($s$ is incomplete)
        put(worklist,$s$)
      manage_targets($s$)
```

Fig. 1. The architecture of the Otter symbolic execution engine

state from the worklist via pick and symbolically executes the next instruction by calling step. As Otter executes instructions, it may encounter conditionals whose guards depend on symbolic values. At these points, Otter queries STP [10], an SMT solver, to see if legal, concrete representations of the symbolic values could make either or both branches possible, and whether an error such as an assertion failure may occur. The symbolic executor will return these states to the scheduler, and those that are *incomplete* (i.e., non-terminal) are added back to the worklist. The call to manage_targets is just for guiding CCBSE's backward search (it is a no-op for other strategies), and is discussed further below.

2.1 Forward Symbolic Execution

Different forward symbolic execution strategies are distinguished by their implementation of the pick function. In Otter we have implemented, among others, three search strategies described in the literature:

Random Path (RP) [3,4] is a probabilistic version of breadth-first search. RP randomly chooses from the worklist states, weighing a state with a path of length n by 2^{-n}. Thus, this approach favors shorter paths, but treats all paths of the same length equally.

KLEE [4] uses a round-robin of RP and what we call *closest-to-uncovered*, which computes the distance between the end of each state's path and the closest uncovered node in the interprocedural control-flow graph and then randomly chooses from the set of states weighed inversely by distance. To our knowledge, KLEE's algorithm has not been described in detail in the literature; we studied it by examining KLEE's source code [18].

SAGE [13] uses a coverage-guided *generational* search to explore states in the execution tree. At first, SAGE runs the initial state until the program terminates by randomly choosing a state to run whenever the symbolic execution core returns multiple states. It stores the remaining states into the worklist as the *first generation children*. Next, SAGE runs each of the first generation children to completion, in the same manner as the initial state, but separately grouping the grandchildren by their first generation parent. After exploring the first generation, SAGE explores subsequent generations (children of the first generation, grandchildren of the first generation, etc.) in a more intermixed fashion, using a block coverage heuristic to determine which generations to explore first.

```
1  int main(void) {
2    int argc; char argv[MAX_ARGC][1];
3    symbolic(&argc); symbolic(&argv);
4    int i, n = 0, b[4] = { 0, 0, 0, 0 };
5    for (i = 0; i < argc; i++) {
6      if (*argv[i] == 'b') {
7        assert(n < 4);
8        b[n++] = 1; /* potential buf. overflow */
9      } else
10       foo(); /* some expensive function */
11   }
12   while (1) {
13     if (getchar()) /* get symbolic input */
14       /* ...do something... */;
15   }
16   return 0;
17 }
```

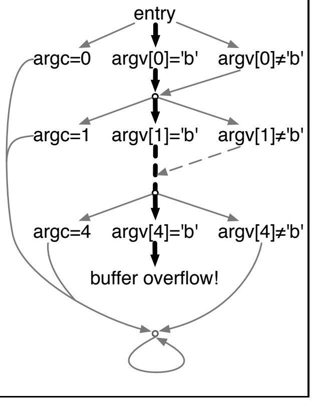


Fig. 2. Example illustrating SDSE's potential benefit

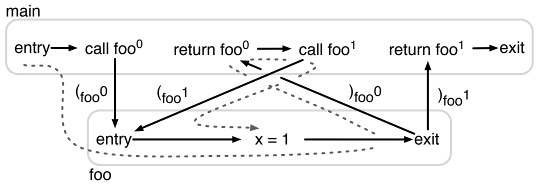
2.2 Shortest-Distance Symbolic Execution

The basic idea of SDSE is to prioritize program branches that correspond to the shortest path-to-target in the ICFG. To illustrate how SDSE works, consider the code in Figure 2, which performs command-line argument processing followed by some program logic, a pattern common to many programs. This program first enters a loop that iterates up to argc times, processing the i$^{\text{th}}$ command-line argument in argv during iteration i. If the argument is 'b', the program sets b[n] to 1 and increments n (line 8); otherwise, the program calls foo. A potential buffer overflow could occur at line 8 when more than four arguments are 'b'; we add an assertion on line 7 to identify when this overflow would occur. After the arguments are processed, the program enters a loop that reads and processes character inputs (lines 12 onward).

Suppose we would like to reason about a possible failure of the assertion. Then we can run this program with symbolic inputs, which we identify with the calls on line 3 to the special built-in function symbolic. The right half of the figure illustrates the possible program paths the symbolic executor can explore on the first five iterations of the argument-processing loop. Notice that for five loop iterations there is only one path that reaches the failing assertion out of $\sum_{n=0}^{4} 3 \times 2^n = 93$ total paths. Moreover, the assertion is not reachable once exploration has advanced past the argument-processing loop.

In this example, RP would have only a small chance of finding the overflow, spending most of its time exploring paths shorter than the one that leads to the buffer overflow. A symbolic executor using KLEE or SAGE would focus on increasing coverage to all lines, wasting significant time exploring paths through the loop at the end of the program, which does not influence this buffer overflow.

In contrast, SDSE works very well in this example, with line 7 set as the target. Consider the first iteration of the loop. The symbolic executor will branch upon reaching the loop guard, and will choose to execute the first instruction of



(a) Example *PN*-path in an interprocedural CFG.

$$\begin{array}{rcl} PN & \rightarrow & P\ N \\ P & \rightarrow & S\ P \\ & | &)_i\ P \\ & | & \epsilon \end{array} \qquad \begin{array}{rcl} N & \rightarrow & S\ N \\ & | & (_i\ N \\ & | & \epsilon \\ S & \rightarrow & (_i\ S\ )_i \\ & | & S\ S \\ & | & \epsilon \end{array}$$

(b) Grammar of *PN* paths.

Fig. 3. SDSE distance computation

the loop, which is two lines away from the assertion, rather than the first instruction after the loop, which can no longer reach the assertion. Next, on line 6, the symbolic executor takes the true branch, since that reaches the assertion itself immediately. Then, determining that the assertion is true, it will run the next line, since it is only three lines away from the assertion and hence closer than paths that go through foo (which were deferred by the choice to go to the assertion). Then the symbolic executor will return to the loop entry, repeating the same process for subsequent iterations. As a result, SDSE explores the central path shown in bold in the figure, and thereby quickly find the assertion failure.

Implementation. SDSE is implemented as a pick function from Figure 1. As mentioned, SDSE chooses the state on the worklist with the shortest *distance to target*. Within a function, the distance is just the number of edges between statements in the control flow graph (CFG). To measure distances across function calls we count edges in an interprocedural control-flow graph (ICFG) [21], in which function call sites are split into *call nodes* and *return nodes*, with *call edges* connecting call nodes to function entries and *return edges* connecting function exits to return nodes. For each call site i, we label call and return edges by $(_i$ and $)_i$, respectively. Figure 3(a) shows an example ICFG for a program in which main calls foo twice; here call i to foo is labeled foo^i.

We define the distance-to-target metric to be the length of the shortest path in the ICFG from an instruction to the target, such that the path contains no mismatched calls and returns. Formally, we can define such paths as those whose sequence of edge labels form a string produced from the *PN* nonterminal in the grammar shown in Figure 3(b). In this grammar, developed by Reps [32] and later named by Fähndrich et al [9,30], S-paths correspond to those that exactly match calls and returns; N-paths correspond to entering functions only; and P-paths correspond to exiting functions only. For example, the dotted path in Figure 3(a) is a *PN*-path: it traverses the matching $(_{foo^0}$ and $)_{foo^0}$ edges, and then traverses $(_{foo^1}$ to the target. Notice that we avoid conflating edges of different call sites by matching $(_i$ and $)_i$ edges, and thus we can statically compute a context-sensitive distance-to-target metric.

PN-reachability was previously used for conservative static analysis [9,30,19]. However, in SDSE, we are always asking about *PN*-reachability from the current instruction. Hence, rather than solve reachability for an arbitrary initial P-path segment (which would correspond to asking about distances from the current

```
1  void main() {
2    int m, n, i;
3    symbolic(&m, sizeof(m), "m");
4    symbolic(&n, sizeof(n), "n");
5
6    for (i=0;i<1000;i++)
7      if (m == i) f(m, n);
8  }
```

```
10 void f(int m, int n) {
11   int i, a, sum=0;
12   for (i=0;i<6;i++) {
13     a = n%2;
14     if (a) sum += a+1;
15     n/=2;
16   }
17   while(1) {
18     if (sum==0 && m==7)
19       assert(0);
20   }
21 }
```

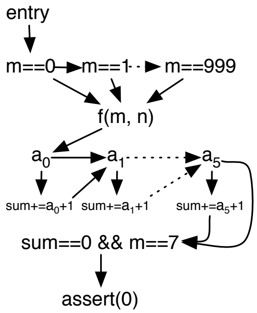


Fig. 4. Example illustrating CCBSE's potential benefit

instruction in all calling contexts of that instruction), we restrict the initial P-path segment to the functions on the current call stack. For performance, we statically pre-compute N-path and S-path distances for all instructions to the target and combine them with P-path distances on demand.

2.3 Call-Chain-Backward Symbolic Execution

SDSE is often very effective, but there are cases on which it does not do well—in particular, SDSE is less effective when there are many potential paths to the target line, but there are only a few, long paths that are realizable. In these situations, CCBSE can sometimes work dramatically better.

To see why, consider the code in Figure 4. This program initializes m and n to be symbolic and then loops, calling f(m, n) when m == i for i $\in [0, 1000)$. For non-negative values of n, the loop in lines 12–16 iterates through n's least significant bits (stored in a during iteration), incrementing sum by a+1 for each non-zero a. Finally, if sum == 0 and m == 7, the failing assertion on line 19 is reached. Otherwise, the program falls into an infinite loop, as sum and m are never updated in the loop.

RP, KLEE, SAGE, and SDSE all perform poorly on this example. SDSE gets stuck at the very beginning: in main's for-loop, it immediately steps into f when m == 0, as this is the "fastest" way to reach the assertion inside f according to the ICFG. Unfortunately, the guard of the assertion is never satisfied when m is 0, and therefore SDSE gets stuck in the infinite loop. SAGE is very likely to get stuck, because the chance of SAGE's first generation entering f with the right argument (m == 7) is extremely low, and SAGE always runs its first generation to completion, and hence will execute the infinite loop forever. RP and KLEE will also reach the assertion very slowly, since they waste time executing f where m$\neq$ 7; none of these paths lead to the assertion failure.

In contrast, CCBSE begins by running f with both parameters m and n set to symbolic, as CCBSE does not know what values might be passed to f. Hence, CCBSE will potentially explore all 2^6 paths induced by the for loop, and one of them, say p, will reach the assertion. When p is found, CCBSE will jump to main

```
manage_targets ($s$)
  (sf,$p$) = path($s$)
  if pc($p$) $\in$ targets
    update_paths(sf, $p$)
  else if pc($p$) = callto(f) and has_paths(f)
    for $p' \in$ get_paths(f)
      if ($p + p'$ feasible)
        update_paths(sf, $p + p'$)

update_paths (sf, $p$)
  if not(has_paths(sf))
    add_callers(sf,worklist)
  add_path(sf, $p$);
```

Fig. 5. Target management for CCBSE

and explore various paths that reach the call to f. At the call to f, CCBSE will follow p to short-circuit the evaluation through f (in particular, the 2^6 branches induced by the for-loop), and thus quickly find a realizable path to the failure.

Implementation. CCBSE is implemented in the manage_targets and pick functions from Figure 1. Otter states s, returned by pick, include the function f in which symbolic execution started, which we call the *origin function*. Thus, traditional symbolic execution states always have main as their origin function, while CCBSE allows different origin functions. In particular, CCBSE begins by initializing states for functions containing target lines.

To start symbolic execution at an arbitrary function Otter must initialize symbolic values for the function's inputs (parameters and global variables). Integer-valued inputs are initialized to symbolic words, and pointers are represented using *conditional pointers*, manipulated using Morris's general axiom of assignment [1,26]. To support recursive data structures, Otter initializes pointers lazily—we do not actually create conditional pointers until a pointer is used, and we only initialize as much of the memory map as is required. When initialized, pointers are set up as follows: for inputs p of type *pointer to type T*, we construct a conditional pointer such that p may be null or p may point to a fresh symbolic value of type T. If T is a primitive type, we also add a disjunct in which p may point to any element of an array of 4 fresh values of type T. This last case models parameters that are pointers to arrays, and we restrict its use to primitive types for performance reasons. In our experiments, we have not found this restriction to be problematic. This strategy for initializing pointers is unsound in that CCBSE could miss some targets, but final paths CCBSE produces are always feasible since they ultimately connect back to main.

The pick function works in two steps. First, it selects the origin function to execute and then it selects a state with that origin. For the former, it picks the function f with the shortest-length call chain from main. For non-CCBSE the origin will always be main. At the start of CCBSE with a single target, the origin will be the one containing the target; as execution continues there will be more choices—picking the "shortest to main" ensures that we move backward from target functions toward main. After selecting the origin function f, pick chooses

```
1  void main() {
2    int m, n;
3    symbolic(&m, sizeof(m), "m");
4    symbolic(&n, sizeof(n), "n");
5    foo(); // Some work
6    if (m >= 30) g(m, n);
7  }
8  void g(int m, int n) {
9    int i;
10   for (i=0;i<1000;i++) {
11     if (m == i) f(m, n);
12   }
13 }
14 void f(int m, int n) {
15   int i, a, sum=0;
16   for (i=0;i<6;i++) {
17     a = n%2;
18     if (a) sum += a+1;
19     n/=2;
20   }
21   while (1) {
22     if (sum==0 && m==37)
23       assert(0);
24   }
25 }
```

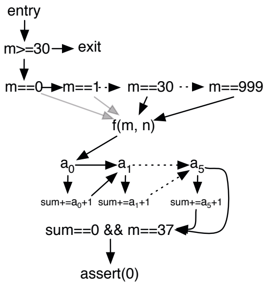


Fig. 6. Example illustrating Mix-CCBSE's potential benefit

one of f's states according to some forward search strategy. We write CCBSE(S) to denote CCBSE using forward search strategy S.

The manage_targets(s) function is given in Figure 5. Recall from Figure 1 that s has already been added to the worklist for additional, standard forward search; the job of manage_targets is to record which paths reach the target line and to try to connect s with path suffixes previously found to reach the target. The manage_targets function extracts from s both the origin function sf and the (interprocedural) *path* p that has been explored from sf to the current point. This path contains all the decisions made by the symbolic executor at condition points. If path p's end (denoted pc(p)) has reached a target (line 10), we associate p with sf by calling update_paths; for the moment one can think of this function as adding p to a list of paths that start at sf and reach targets. Otherwise, if the path's end is at a call to some function f, and f itself has paths to targets, then we may possibly extend p with one or more of those paths. So we retrieve f's paths, and for each one p' we see whether concatenating p to p' (written $p + p'$) produces a feasible path. If so, we add it to sf's paths. Feasibility is checked by attempting to symbolically execute p' starting in p's state s.

Now we turn to the implementation of update_paths. This function simply adds p to sf's paths (line 19), and if sf did not previously have any paths, it will create initial states for each of sf's callers (pre-computed from the call graph) and add these to the worklist (line 17). Because these callers will be closer to main, they will be subsequently favored by pick when it chooses states.

2.4 Mixing CCBSE with Forward Search

While CCBSE may find a path more quickly, it comes with a cost: its queries tend to be more complex than in forward search, and it can spend significant time trying paths that start in the middle of the program but are ultimately infeasible. Consider Figure 6, a modified version of the code in Figure 4. Here, main calls function g, which acts as main did in Figure 4, with some m >= 30 (line 6), and the assertion in f is reachable only when m == 37 (line 22). All other strategies fail in the same manner as they do in Figure 4.

However, CCBSE also fails to perform well here, as it does not realize that m is at least 30, and therefore considers ultimately infeasible conditions $0 \leq \texttt{m} \leq 36$ in f. With Mix-CCBSE, however, we conceptually start forward symbolic execution from main at the same time that CCBSE ("backward search") is run. As before, the backward search will gets stuck in finding a path from g's entry to the assertion. However, in the forward search, g is called with $\texttt{m} \geq 30$, and therefore f is always called with $\texttt{m} \geq 30$, making it hit the right condition m == 37 very soon thereafter. Notice that, in this example, the backward search must find the path from f's entry to the assertion *before* f is called with m == 37 in the forward search in order for the two searches to match up (e.g., there are enough instructions to run in line 5). Should this not happen, Mix-CCBSE degenerates to its constituents running independently in parallel, which is the worst case.

Implementation. We implement Mix-CCBSE with a slight alteration to pick. At each step, we decide whether to use regular forward search or CCBSE next, splitting the strategies 50/50 by time spent. We compute time heuristically as $50 \times (\text{no. of solver calls}) + (\text{no. of instructions executed})$, taking into account the higher cost of solver queries over instruction executions.[1]

3 Experiments

We evaluated our directed search strategies by comparing their performance on the small example programs from Section 2 and on bugs reported in six programs from GNU Coreutils version 6.10. These bugs were previously discovered by KLEE [4]. All experiments were run on a machine with six 2.4Ghz quad-core Xeon E7450 processors and 48GB of memory, running 64-bit Linux 2.6.26. We ran 16 tests in parallel, observing minimal resource contention. The tests required less than 2 days of elapsed time. Total memory usage was below 1GB per test.

The results are presented in Table 1. Part (a) of the table gives the results for our directed search strategies. For comparison, we also implemented an intraprocedural variant of SDSE that ignores call-chains: if the target is not in the current function, then the distance-to-target is ∞. We refer to the intraprocedural variant as IntraSDSE, and to standard SDSE as InterSDSE. This table lists three variants of CCBSE, using RP, InterSDSE, or IntraSDSE as the forward strategy. In the last two cases, we modified Inter- and IntraSDSE slightly to compute shortest distances to the target line *or* to the functions reached in CCBSE's backward search. This allows those strategies to take better advantage of CCBSE (otherwise they would ignore CCBSE's search in determining which paths to take).

Part (b) of the table gives the results from running KLEE version r130848 [18], and part (c) gives the results for forward search strategies implemented in Otter, both by themselves and mixed with CCBSE(RP). We chose CCBSE(RP) because it was the best overall of the three from part (a), and because RP is the fastest

[1] We could also use wall-clock time, however, this leads to non-deterministic, non-reproducible results. We opted to use our heuristic for reproducibility.

Table 1. Statistics from benchmark runs. For each Coreutils program and for the total, the fastest two times are highlighted. Key: Median SIQR(Outliers) ∞ : time out

	Inter-SDSE	Intra-SDSE	CCBSE(*X*) where *X* is RP	InterSDSE	IntraSDSE
Figure 2	0.4 0.0	0.4 0.0(5)	16.2 2.4(6)	0.5 0.0(1)	0.4 0.0(3)
Figure 4	∞	∞	60.8 7.8(4)	7.3 1.2(3)	7.2 1.0(4)
Figure 6	∞	∞	∞	∞	∞
mkdir	34.7 19.7(10)	∞	163.0 42.5	150.3 93.4	150.7 93.9
mkfifo	13.1 0.4	∞	70.2 17.3	49.7 21.8	49.3 23.2(1)
mknod	∞	∞	216.5 60.7	∞	∞
paste	12.6 0.5	56.4 5.4	26.0 0.5(1)	31.0 4.8	32.1 4.0
ptx	18.4 0.6(4)	103.5 19.7(1)	24.2 0.7(1)	24.5 0.9(3)	24.1 1.1(2)
seq	12.1 0.4(1)	∞	30.9 1.4	369.3 425.9(6)	391.8 411.1(6)
Total	1891.0	7360.0	530.9	2424.8	2448.0

(a) Directed search strategies

	KLEE
Figure 2	2.6 0.0(7)
Figure 4	∞
Figure 6	∞
mkdir	∞
mkfifo	274.2 315.6(9)
mknod	851.6 554.2(8)
paste	30.6 9.7(8)
ptx	93.8 81.7(7)
seq	38.2 14.5(8)
Total	3088.55

(b) KLEE

	Otter-KLEE		Otter-SAGE		Random Path	
	Pure	w/CCBSE	Pure	w/CCBSE	Pure	w/CCBSE
Figure 2	101.1 57.5(4)	104.8 57.3(5)	∞	∞	15.3 2.2(6)	16.1 2.6(6)
Figure 4	579.7 ∞	205.5 133.1(9)	∞	∞	160.1 6.4(11)	80.6 177.2(9)
Figure 6	587.8 ∞	147.6 62.6(7)	∞	∞	169.8 9.1(8)	106.8 11.2(4)
mkdir	168.9 31.0	124.7 12.1(2)	365.3 354.2(5)	1667.7 ∞	143.5 5.3	136.4 7.9
mkfifo	41.7 5.2(1)	38.2 4.6	77.6 101.1(2)	251.9 257.0(8)	59.4 3.7	52.7 1.8(1)
mknod	174.8 24.1	93.1 12.7	108.5 158.7(5)	236.4 215.0(5)	196.7 3.9(2)	148.9 11.8
paste	22.6 0.5(4)	28.6 0.9(3)	54.9 36.2(5)	60.4 52.1(3)	22.1 0.6	27.3 1.0(1)
ptx	33.2 3.9	27.1 2.7	∞	∞	28.9 0.8	28.1 1.1(2)
seq	354.8 94.3(1)	49.3 5.1(1)	∞	288.8 ∞	170.8 3.7(3)	35.9 1.4(1)
Total	795.8	360.9	4206.4	4305.3	621.3	429.4

(c) Undirected search strategies and their mixes with CCBSE(RP)

of the forward-only strategies in part (c). We write Mix-CCBSE(S) to denote the mixed strategy where S is the forward search strategy and CCBSE(RP) is the backward strategy. We did not directly compare against execution synthesis (ESD) [36], a previously proposed directed search strategy; at the end of this section we relate our results to those reported in the ESD paper.

We found that the randomness inherent in most search strategies and in the STP theorem prover introduces tremendous variability in the results. Thus, we ran each strategy/target condition 41 times, using integers 1 to 41 as random seeds for Otter. (We were unable to find a similar option in KLEE, and so simply ran it 41 times.) The main numbers in Table 1 are the medians of these runs, and the small numbers are the semi-interquartile range (SIQR). The number of outliers—which fall 3×SIQR below the lower quartile or above the upper quartile, if non-zero—is given in parentheses. We ran each test for at most 600 seconds for the synthetic examples, and at most 1,800 seconds for the Coreutils programs. The median is ∞ if more than half the runs timed out, while the SIQR is ∞ if more than one quarter of the runs timed out. We highlight the fastest two times in each row.

3.1 Synthetic Programs

The first three rows in Table 1 give the results from the examples in Figures 2, 4, and 6. In all cases the programs behaved as predicted.

For the program in Figure 2, both InterSDSE and IntraSDSE performed very well. Since the target line is in main, CCBSE(*SDSE) is equivalent to *SDSE, so those variants performed equally well. Otter-KLEE took much longer to find the target, whereas Otter-SAGE timed out for more than half the runs. RP was able to find the target, but it took much longer than *SDSE. Note that CCBSE(RP) degenerates to RP in this example, and runs in about the same time as RP. Lastly, KLEE performed very well also, although it was still slower than *SDSE in this example.

For the program in Figure 4, CCBSE(InterSDSE) and CCBSE(IntraSDSE) found the target line quickly, while CCBSE(RP) did so in reasonable amount of time. CCBSE(*SDSE) were much more efficient, because with these strategies, after each failing verification of f(m,n) (when $0 \leq$ m < 7), the *SDSE strategies chose to try f(m+1,n) rather than stepping into f, as f is a target added by CCBSE and is closer from any point in main than the assertion in f is.

For the program in Figure 6, Mix-CCBSE(RP) and Mix-CCBSE(Otter-KLEE) performed the best among all strategies, as expected. However, Mix-CCBSE (Otter-SAGE) performed far worse. This is because its forward search (Otter-SAGE) got stuck in one value of m in the very beginning, and therefore it and the backward search did not match up.

3.2 GNU Coreutils

The lower rows of Table 1 give the results from the Coreutils programs. The six programs we analyzed contain a total of 2.4 kloc and share a common library of about 30 kloc. For each bug, we manually added a corresponding failing assertion to the program, and set that as the target line. For example, the Coreutils program seq has a buffer overflow in which an index i accesses outside the bounds of a string fmt [25]. Thus, just before this array access, we added an assertion **assert**(i<strlen(fmt)) to indicate the overflow. Each assertion has a call-chain distance from main ranging from two to seven. Note that Otter does have built-in detection of buffer overflows and similar errors, but for our experiments we do not use this feature to identify valid targets for line reachability.

The Coreutils programs receive input from the command line and from standard input. We initialized the command line as in KLEE [4]: given a sequence of integers $n_1, n_2, \cdots, n_k$, Otter sets the program to have (excluding the program name) at least 0 and at most k arguments, where the ith argument is a symbolic string of length n_i. All of the programs we analyzed used $(10, 2, 2)$ as the input sequence, except mknod, which used $(10, 2, 2, 2)$. Standard input is modeled as an unbounded stream of symbolic values.

```
int main(int argc, char** argv) {
    while ((optc = getopt_long (argc, argv, opts, longopts, NULL)) != −1) { ... } ...
    if (/* some condition */) quote(...);
    ...
    if (/* another condition */) quote(...);
}
```

Fig. 7. Code pattern in mkdir, mkfifo and mknod

Coreutils programs make extensive use of the C standard library. To support them, we implemented a partial model of POSIX system calls on top of an in-memory file system, and combined this with the newlib C standard library implementation [28]. All this code is written in C, so Otter executes it as it would any other source code.

Analysis. We can see clearly from the shaded boxes in Table 1 that InterSDSE performed extremely well, achieving the fastest running times on five of the six programs. However, InterSDSE timed out on mknod. Examining this program, we found it shares a similar structure with mkdir and mkfifo, sketched in Figure 7. These programs parse their command line arguments with getopt_long, and then branch depending on those arguments; several of these branches call the same function quote(). In mkdir and mkfifo, the target is reachable within the first call to quote(), and thus SDSE can find it quickly. However, in mknod, the bug is only reachable in a later call to quote()—but since the first call to quote() is a shorter path to the target line, InterSDSE takes that call and then gets stuck inside quote(), never returning to main() to find the path to the failing assertion.

The last row in Table 1 sums up the median times for the Coreutils programs, counting time-outs as 1,800s. These results show that mixing a forward search with CCBSE can be a significant improvement—for Otter-KLEE and Random Path, the total times are notably less when mixed with CCBSE. One particularly interesting result is that Mix-CCBSE(Otter-KLEE) runs dramatically faster on mknod than either of its constituents (93.1s for the combination versus 174.8s for Otter-KLEE and 216.5s for CCBSE(RP)). This case demonstrates the benefit of mixing forward and backward search: in the combination, CCBSE(RP) found the failing path inside of quote() (recall Figure 7), and Otter-KLEE found the path from the beginning of main() to the right call to quote(). We also observe that the SIQR for Mix-CCBSE(Otter-KLEE) is generally lower than either of its constituents, which is a further benefit.

Overall, Mix-CCBSE(Otter-KLEE) has the fastest total running time across all strategies, including InterSDSE (because of its time-out); and although it is not always the fastest search strategy, it is subjectively fast enough on these examples. Thus, our results suggest that the best single strategy option for solving line reachability is Mix-CCBSE(Otter-KLEE), or perhaps Mix-CCBSE(Otter-KLEE) in round-robin with InterSDSE to combine the strengths of both.

Execution Synthesis. ESD [36] is a symbolic execution tool that also aims to solve the line reachability problem. It uses a *proximity-guided path search* that is similar to our *Intra*SDSE algorithm, and an interprocedural reaching definition analysis to find intermediate goals for directing the search. The published results show that ESD works very well on five Coreutils programs, four of which (mkdir, mkfifo, mknod, and paste) we also analyzed. Since ESD is not publicly available, we were unable to include it in our experiment directly, and we found it difficult to replicate from the description in the paper. One thing we can say for certain is that the interprocedural reaching definition analysis in ESD is clearly critical, as our implementation of IntraSDSE by itself performed quite poorly.

Comparing published numbers, if we run InterSDSE and Mix-CCBSE(Otter-KLEE) simultaneously on two machines and return whichever returns first, we obtain a strategy which performs in the same ballpark as ESD, which took 15s for mkdir, 15s for mkfifo, 20s for mknod, and 25s for paste. The ESD authors informed us that they did not observe variability in their experiment, which consists of 5 runs per test program [35]. However, we find this somewhat surprising, since ESD employs randomization in its search strategy, and is implemented on top of KLEE whose performance we have found to be highly variable (Table 1).

Clearly this comparison should be taken with a grain of salt due to major differences between Otter and ESD as well as in the experimental setups. These include the version of KLEE evaluated (we used the latest version of KLEE as of April 2011, whereas the ESD paper is based on a pre-release 2008 version of KLEE), symbolic parameters, default search strategy, processor speed, memory, Linux kernel version, whether tests are run in parallel or sequentially, the number of runs per test program, and how random number generators are seeded. These differences may also explain a discrepancy between our evaluations of KLEE: the ESD paper reported that KLEE was not able to find the target bugs within an hour, but in our experiments KLEE was able to find them (note that nearly one-third of the runs for mkdir returned within half an hour, which is not reflected by its median).

3.3 Threats to Validity

There are several threats to the validity of our results. First, we were surprised by the wide variability in our running times: the SIQR can be very large—in some cases for CCBSE(*SDSE), KLEE and Otter-SAGE, the SIQR exceeds the median—and there are many outliers.[2] This indicates the results are not normally distributed, and suggests that randomness in symbolic execution can greatly perturb the results. To our knowledge, this kind of significant variability has not been reported well in the literature, and we recommend that future efforts on symbolic execution carefully consider it in their analyses. That said, the variation in results for CCBSE(Otter-KLEE) and InterSDSE, the best-performing strategies, was generally low.

Second, our implementation of KLEE and SAGE unavoidably differs from the original versions. The original KLEE is based on LLVM [22], whereas Otter is based on CIL, and therefore they compute distance metrics over different control-flow graphs. Also, Otter uses newlib [28] as the standard C library, while

[2] See the companion technical report, Appendix A for beeswarm distribution plots for each cell in the table [23].

KLEE uses uclibc [33]. These may explain some of the difference between KLEE and Otter-KLEE's performance in Table 1.

Finally, the original SAGE is a concolic executor, which runs programs to completion using the underlying operating system, while Otter-SAGE emulates the run-to-completion behavior by not switching away from the currently executing path. There are other differences between SAGE and Otter, e.g., SAGE only invokes the theorem prover at the end of path exploration, whereas Otter invokes the theorem prover at every conditional along the path. Also, SAGE suffers from *divergences*, where a generated input may not follow a predicted path (possibly repeating a previously explored path) due to mismatches between the system model and the underlying system. Otter does not suffer from divergences because it uses a purely symbolic system model. These differences may make the SAGE strategy less suited to Otter.

4 Other Related Work

Several other researchers have proposed general symbolic execution search strategies, in addition to the ones discussed in Section 2. Hybrid concolic testing mixes random testing with symbolic execution [24]. Burnim and Sen propose several such heuristics, including a control-flow graph based search strategy [3]. Xie et al propose Fitnex, a strategy that uses fitness values to guide path exploration [34]. It would be interesting future work to compare against these strategies as well; we conjecture that, as these are general rather than targeted search strategies, they will not perform as well as our approach for targeted search.

Researchers have also used model checkers to solve the line reachability problem by specifying the target line as the target state in the model. Much like our work, *directed model checking* [7] focuses on scheduling heuristics to quickly discover the target. Edelkamp et al proposed several heuristics based on minimizing the number of transitions from the current program state to the target state in the model defined by a finite-state automata [8] or Büchi automata [7]. Groce et al suggested using *structural heuristics* such as maximizing code coverage or thread interleavings [14]. Kupferschmid et al borrowed an AI technique based on finding the shortest distance through a *monotonic relaxation* of the model in which states are sets whose successors increase monotonically under set inclusion [20]. In contrast, SDSE prioritizes exploration based on distance in the ICFG, and CCBSE explores backward from the target.

5 Conclusion

In this paper, we studied the problem of line reachability, which arises in automated debugging and in triaging static analysis results, among other applications. We introduced two new directed search strategies, SDSE and CCBSE, that use two very different approaches to solve line reachability. We also discussed a method for combining CCBSE with any forward search strategy, to get the best of both worlds. We implemented these strategies and a range of state-of-the-art forward search strategies (KLEE, SAGE, and Random Path) in Otter, and studied their performance on six programs from GNU Coreutils and on three synthetic programs. The results indicate that both SDSE and mixed CCBSE and KLEE outperformed the other strategies. While SDSE performed extremely well in many cases, it does perform badly sometimes, whereas mixing CCBSE with KLEE achieves the best overall running time across all strategies, including SDSE. In summary, our results suggest that directed symbolic execution is a practical and effective approach to line reachability.

Acknowledgments. This research was supported in part by National Science Foundation grants CCF-0346982, CCF-0541036, and CCF-0915978. We would also like to thank Elnatan Reisner and Jonathan Turpie for their help developing the POSIX library.

References

1. Bornat, R.: Proving pointer programs in Hoare logic. In: MPC, pp. 102–126 (2000)
2. Boyer, R.S., Elspas, B., Levitt, K.N.: SELECT–a formal system for testing and debugging programs by symbolic execution. In: ICRS, pp. 234–245 (1975)
3. Burnim, J., Sen, K.: Heuristics for scalable dynamic test generation. In: ASE, pp. 443–446 (2008)
4. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI, pp. 209–224 (2008)
5. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: automatically generating inputs of death. In: CCS, pp. 322–335 (2006)
6. Coreutils - GNU core utilities, http://www.gnu.org/software/coreutils/
7. Edelkamp, S., Leue, S., Lluch-Lafuente, A.: Directed explicit-state model checking in the validation of communication protocols. Software Tools for Technology Transfer 5(2), 247–267 (2004)
8. Edelkamp, S., Lluch-Lafuente, A., Leue, S.: Trail-directed model checking. Electrical Notes Theoretical Computer Science 55(3), 343–356 (2001)
9. Fähndrich, M., Rehof, J., Das, M.: Scalable context-sensitive flow analysis using instantiation constraints. In: PLDI, pp. 253–263 (2000)
10. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 519–531. Springer, Heidelberg (2007)
11. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: PLDI, pp. 213–223 (2005)
12. Godefroid, P., Levin, M.Y., Molnar, D.A.: Active property checking. In: EMSOFT, pp. 207–216 (2008)
13. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: NDSS (2008)
14. Groce, A., Visser, W.: Model checking Java programs using structural heuristics. In: ISSTA, pp. 12–21 (2002)
15. Howden, W.E.: Symbolic testing and the DISSECT symbolic evaluation system. IEEE Transactions on Software Engineering 3(4), 266–278 (1977)
16. Khoo, Y.P., Chang, B.-Y.E., Foster, J.S.: Mixing type checking and symbolic execution. In: PLDI, pp. 436–447 (2010)
17. King, J.C.: Symbolic execution and program testing. CACM 19(7), 385–394 (1976)

18. The KLEE Symbolic Virtual Machine, http://klee.llvm.org
19. Kodumal, J., Aiken, A.: The set constraint/CFL reachability connection in practice. In: PLDI, pp. 207–218 (2004)
20. Kupferschmid, S., Hoffmann, J., Dierks, H., Behrmann, G.: Adapting an AI planning heuristic for directed model checking. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 35–52. Springer, Heidelberg (2006)
21. Landi, W., Ryder, B.G.: Pointer-induced aliasing: a problem taxonomy. In: POPL, pp. 93–103 (1991)
22. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis transformation. In: CGO, pp. 75–86 (2004)
23. Ma, K.-K., Khoo, Y.P., Foster, J.S., Hicks, M.: Directed symbolic execution. Technical Report CS-TR-4979, UMD-College Park (April 2011)
24. Majumdar, R., Sen, K.: Hybrid concolic testing. In: ICSE, pp. 416–426 (2007)
25. Meyering, J.: Seq: give a proper diagnostic for an invalid –format=% option (2008), http://git.savannah.gnu.org/cgit/coreutils.git/commit/?id=b8108fd2ddf77ae79cd014f4f37798a52be13fd1
26. Morris, J.M.: A general axiom of assignment. Assignment and linked data structure. A proof of the Schorr-Waite algorithm. In: Broy, M., Schmidt, G. (eds.) Theoretical Foundations of Programming Methodology, pp. 25–51 (1982)
27. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: CC 2002. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002)
28. The Newlib Homepage, http://sourceware.org/newlib/
29. Osterweil, L.J., Fosdick, L.D.: Program testing techniques using simulated execution. In: ANSS, pp. 171–177 (1976)
30. Rehof, J., Fähndrich, M.: Type-base flow analysis: from polymorphic subtyping to CFL-reachability. In: PLDI, pp. 54–66 (2001)
31. Reisner, E., Song, C., Ma, K.-K., Foster, J.S., Porter, A.: Using symbolic evaluation to understand behavior in configurable software systems. In: ICSE, pp. 445–454 (2010)
32. Reps, T.W.: Program analysis via graph reachability. In: ILPS, pp. 5–19 (1997)
33. μClibc, http://www.uclibc.org/
34. Xie, T., Tillmann, N., de Halleux, J., Schulte, W.: Fitness-guided path exploration in dynamic symbolic execution. In: DSN, pp. 359–368 (2009)
35. Zamfir, C.: Personal communication (May 2011)
36. Zamfir, C., Candea, G.: Execution synthesis: a technique for automated software debugging. In: EuroSys, pp. 321–334 (2010)

Statically Validating Must Summaries for Incremental Compositional Dynamic Test Generation

Patrice Godefroid[1], Shuvendu K. Lahiri[1], and Cindy Rubio-González[2]

[1] Microsoft Research, Redmond, WA, USA
[2] University of Wisconsin, Madison, WI, USA

Abstract. Compositional dynamic test generation can achieve significant scalability by memoizing symbolic execution sub-paths as test summaries. In this paper, we formulate the problem of statically validating symbolic test summaries against code changes. Summaries that can be proved still valid using a static analysis of a new program version do not need to be retested or recomputed dynamically. In the presence of small code changes, incrementality can considerably speed up regression testing since static checking is much cheaper than dynamic checking and testing. We provide several checks ranging from simple syntactic ones to ones that use a theorem prover. We present preliminary experimental results comparing these approaches on three large Windows applications.

1 Introduction

Whitebox fuzzing [15] is a promising new form of security testing based on dynamic test generation [5, 14]. Dynamic test generation consists of running a program while simultaneously executing the program symbolically in order to gather constraints on inputs from conditional statements encountered along the execution. Those constraints are then systematically negated and solved with a constraint solver, generating new test inputs to exercise different execution paths of the program. Over the last couple of years, whitebox fuzzing has extended the scope of dynamic test generation from unit testing to whole-program security testing, thanks to new techniques for handling very long execution traces (with billions of instructions). In the process, whitebox fuzzers have found many new security vulnerabilities (buffer overflows) in Windows [15] and Linux [21] applications, including codecs, image viewers and media players. Notably, our whitebox fuzzer SAGE found roughly one third of *all* the bugs discovered by file fuzzing during the development of Microsoft's Windows 7 [12]. Since 2008, SAGE has been continually running on average 100+ machines automatically "fuzzing" hundreds of applications in a dedicated security testing lab. This represents the largest computational usage ever for any Satisfiability Modulo Theories (SMT) solver [27], according to the authors of the Z3 SMT solver [8].

Despite these successes, several challenges remain, such as increasing code coverage and bug finding, while reducing computational costs. A key promising idea is *compositionality*: the search process can be made compositional by memoizing symbolic execution sub-paths as test summaries which are re-usable during the search, resulting in a search algorithm that can be exponentially faster than a non-compositional one

E. Yahav (Ed.): SAS 2011, LNCS 6887, pp. 112–128, 2011.

[11]. By construction, symbolic test summaries are "must" summaries guaranteeing the existence of some program executions and hence useful for proving *existential* reachability properties (such as the existence of an input leading to the execution of a specific program branch or bug). They dualize traditional "may" summaries used in static program analysis for proving *universal* properties (such as the absence of specific types of bugs for all program paths). We are currently building a general infrastructure to generate, store and re-use symbolic test summaries for large parts of the Windows operating system.

In this context, an important problem is the maintenance of test summaries as the code under test slowly evolves. Recomputing test summaries dynamically from scratch for every program version sounds wasteful, as new versions are frequent and much of the code has typically not changed. Instead, *whenever possible*, it could be *much cheaper* to *statically* check whether previously-computed symbolic test summaries are still valid for the new version of the code. The formalization and study of this problem is the motivation for this paper.

We introduce the *must-summary checking problem*:

> *Given a set S of symbolic test summaries for a program Prog and a new version Prog′ of Prog, which summaries in S are still valid must summaries for Prog′ ?*

We also consider the more general problem of checking whether an arbitrary set S of summaries are valid must summaries for an arbitrary program *Prog*.

We present three algorithms with different precision to statically check which old test summaries are still valid for a new program version. First, we present an algorithm (in Section 3) based on a simple impact analysis of code changes on the static control-flow and call graphs of the program; this algorithm can identify local code paths that have not changed and for which old summaries are therefore still valid. Second, we present (in Section 4) a more precise predicate-sensitive refined algorithm using verification-condition generation and automated theorem proving. Third, we present an algorithm (in Section 5) for checking the validity of a symbolic test summary against a program regardless of program changes, by checking whether the pre/postconditions captured in the old summary still hold on the new program. We discuss the strengths and weaknesses of each solution, and present preliminary experimental results with sample test summaries generated for three large Windows applications. These experiments confirm that hundreds of summaries can be validated statically in minutes, while validating those dynamically can require hours or days.

2 Background and Problem Definition

2.1 Background: Compositional Symbolic Execution

We assume we are given a sequential program *Prog* with input parameters I. Dynamic test generation [14] consists of running the program *Prog* both concretely and symbolically, in order to collect symbolic constraints on inputs obtained from predicates in branch statements along the execution. For each execution path w, i.e., a sequence of statements executed by the program, a *path constraint* ϕ_w is constructed that characterizes the input values for which the program executes along w. Each variable appearing in ϕ_w is thus a program input. Each constraint is expressed in some theory T decided by a constraint solver (for instance, including linear arithmetic, bit-vector operations, etc.). A constraint solver is an automated theorem prover which also returns a satisfying assignment for all variables appearing in constraints it can prove satisfiable. All program paths can be enumerated by a search algorithm that explores all possible branches at conditional statements. The paths w for which ϕ_w is satisfiable are *feasible* and are the only ones that can be executed by the actual program provided the solutions to ϕ_w characterize exactly the inputs that drive the program through w. Assuming that the constraint solver used to check the satisfiability of all formulas ϕ_w is sound and complete, this use of symbolic execution for programs with finitely many paths amounts to program verification.

Systematically testing and symbolically executing *all* feasible program paths does not scale to large programs. Indeed, the number of feasible paths can be exponential in the program size, or even infinite in the presence of loops with unbounded number of iterations. This *path explosion* [11] can be alleviated by performing symbolic execution *compositionally* [2, 11].

Let us assume the program *Prog* consists of a set of functions. In the rest of this section, we use the generic term of *function* to denote any part of the program *Prog* whose observed behaviors are summarized; any program fragments can be treated as "functions" as will be discussed later. To simplify the presentation, we assume the functions in *Prog* do not perform recursive calls, and that all the executions of *Prog* terminate. These assumptions do not prevent *Prog* from having infinitely many executions paths if it contains a loop whose number of iterations depends on some unbounded input.

In compositional symbolic execution, a function summary ϕ_f for a function f is defined as a logic formula over constraints expressed in theory T. ϕ_f can be derived by successive iterations and defined as a *disjunction* of formulas ϕ_{w_f} of the form $\phi_{w_f} = pre_{w_f} \wedge post_{w_f}$, where w_f denotes an intraprocedural path inside f, pre_{w_f} is a conjunction of constraints on the inputs of f, and $post_{w_f}$ is a conjunction of constraints on the outputs of f. An input to a function f is any value that can be read by f, while an output of f is any value written by f. ϕ_{w_f} can be computed automatically from the path constraint for the intraprocedural path w_f [2, 11].

For instance, given the function `is_positive` in Figure 1, a summary ϕ_f for this function can be

$$\phi_f = (x > 0 \wedge ret = 1) \vee (x \leq 0 \wedge ret = 0)$$

where *ret* denotes the value returned by the function.

Symbolic variables are associated with function inputs (like x in the example) and function outputs (like *ret* in the example), in addition to whole-program inputs. In order to generate a new test to cover a new branch b in some function, *all* the previously known summaries can be used to generate a formula ϕ_P representing symbolically all the paths known so far during the search. By construction [11], symbolic variables corresponding to function inputs and outputs are all bound in ϕ_P, and the remaining free variables correspond exclusively to whole-program inputs (since only those can be controlled for test generation).

```
#define N 100
void P(int s[N]) { // N inputs
  int i, cnt = 0;
  for (i = 0; i < N; i++)
    cnt = cnt + is_positive(s[i]);
  if (cnt == 3) error(); // (*)
  return;
}
```

```
int is_positive(int x) {
  if (x > 0) return 1;
  return 0;
}
int g(int x, int y) {
  if ((x > 0)&&(hash(y) > 10))
    return 1;
  return 0;
}
```

Fig. 1. Example

For instance, for the program P in Figure 1, a formula ϕ_P to generate a test covering the then branch (*) given the above summary ϕ_f for function is_positive can be

$$(ret_0 + ret_1 + \ldots + ret_{N-1} = 3) \wedge \bigwedge_{0 \leq i < N} ((s[i] > 0 \wedge ret_i = 1) \vee (s[i] \leq 0 \wedge ret_i = 0))$$

where ret_i denotes the return value of the ith call to function is_positive. Even though program P has 2^{N+1} feasible whole-program paths, compositional test generation can cover "symbolically" all those paths in at most 4 test inputs: 2 tests to cover both branches in function is_positive plus 2 tests to cover both branches of the conditional statement (*). Compositionality avoids an exponential number of tests and calls to the constraint solver, at the cost of using more complex formulas with more disjunctions.

2.2 Problem Definition: Must Summary Checking

In practice, symbolic execution of large programs is bound to be imprecise due to complex program statements (pointer manipulations, floating-point operations, etc.) and calls to operating-system and library functions that are hard to reason about symbolically with good enough precision at a reasonable cost. Whenever precise symbolic execution is not possible during dynamic test generation, concrete values can be used to simplify constraints and carry on with a simplified, partial symbolic execution [14]. The resulting path constraints are then *under-approximate*, and summaries become *must* summaries.

For example, consider the function g in Figure 1 and assume the function hash(y) is a complex or unknown function for which no constraint is generated. Assume we observe at runtime that when g is invoked with $y = 45$, the value of hash(45) is 987. The summary for this execution of function g can then be

$$(x > 0 \wedge y = 45 \wedge ret = 1)$$

Here, symbolic variable y is constrained to be equal to the concrete value 45 observed along the run because the expression $hash(y)$ cannot be symbolically represented. This summary is a must summary since all value pairs (x, y) that satisfy its precondition define executions of g that satisfy the postcondition $ret = 1$. However, this set is a subset of all value pairs that satisfy this postcondition assuming there exists some other value of y different from 45 such that $hash(y) > 10$. For test generation purposes, we safely under-approximate this perfect but unknown input set with the smaller precondition $x > 0 \wedge y = 45$. A must summary can thus be viewed as an abstract witness of some execution. Must summaries are useful for bug finding and test generation, and dualize may summaries for proving correctness, i.e., the absence of bugs.

We denote a must summary by a quadruple $\langle lp, P, lq, Q\rangle$ where lp and lq are arbitrary program locations, P is a summary precondition holding in lp, and Q is a summary postcondition holding in lq. lp and lq can be anywhere in the program: for instance, they can be the entry and exit points of a function (as in the previous examples) or block, or two program points where consecutive symbolic constraints are injected in the path constraint during symbolic execution, possibly in different functions. In what follows, we call a summary *intraprocedural* if its locations *(lp, lq)* are in a same function f and the function f did not return between lp to lq when the summary was generated (i.e., no instruction from a function calling f higher in the call stack was executed from lp to lq when the summary was generated). We will only consider intraprocedural summaries in the remainder of this paper, unless otherwise specified.

Formally, must summaries are defined as follows.

Definition 1. *A must summary $\langle lp, P, lq, Q\rangle$ for a program Prog implies that, for every program state satisfying P at lp in Prog, there exists an execution that visits lq and satisfies Q at lq.*

A must summary is called *valid* for a program *Prog* if it satisfies Definition 1. We define the *must-summary checking problem* as follows.

Definition 2. *(Must-summary checking)* Given a valid must summary $\langle lp, P, lq, Q\rangle$ for a program *Prog* and a new version *Prog′* of *Prog*, is $\langle lp, P, lq, Q\rangle$ still valid for *Prog′*?

We also consider later in Section 5 the more general problem of checking whether an arbitrary must summary is *valid* for an arbitrary program *Prog*. These problems are different from the *must summary inference/generation* problem discussed in prior work [2, 11, 16].

We present three different algorithms for statically checking which old must summaries are still valid for a new program version. These algorithms can be used in isolation or in a pipeline, one after another, in successive "phases" of analysis.

3 Phase 1: Static Change Impact Analysis

The first "Phase 1" algorithm is based on a simple impact analysis of code changes in the static control-flow and call graphs of the program.

A sufficient condition to prove that an old must summary $\langle lp, P, lq, Q\rangle$ generated as described in Section 2.1 is still valid in a new program version is that *all* the instructions that were executed in the original program path taken between lp and lq when the summary was generated remain unchanged in the new program. Recording all unique instructions executed between each pair *(lp, lq)* would be expensive for large programs as many instructions (possibly in other functions) can be executed.

Instead, we can *over-approximate* this set by statically finding all program instructions that *may* be executed on *all* paths from *lp* to *lq*: this solution requires no additional storage of runtime-executed instructions but is less precise. If no instruction in this larger set has changed between the old and new programs, any summary for *(lp, lq)* can then be safely reused for the new program version; otherwise, we have to conservatively declare the summary as potentially invalid since a modified instruction might be on the original path taken from *lp* to *lq* when the summary was generated.

To determine whether a specific instruction in the old program is unchanged in the new program, we rely on an existing lightweight syntactic "diff"-like tool which can (conservatively) identify instructions that have been modified, deleted or added between two program versions by comparing their abstract syntax trees.

Precisely, an instruction i of a program *Prog* is defined as *modified* in another program version $Prog'$ if i is changed or deleted in $Prog'$ or if its ordered set of immediate successor instructions changed between *Prog* and $Prog'$. For instance, swapping the then and else branches of a conditional jump instruction "modifies" the instruction. However, the definition is local as it does not involve non-immediate successors.

Program instructions that are not modified can be mapped across program versions. Conversely, if an instruction cannot be mapped across program versions, it is considered as "deleted" and therefore *modified*. Similarly, a program function is defined as *modified* if it contains either a modified instruction, or a call to a modified function, or a call to an unknown function (e.g., a function outside the program or through a function pointer which we conservatively assume may have been modified). Note that this definition is transitive, unlike the definition of modified instruction.

Given those definitions, we can soundly infer valid summaries using the following rule.

> An intraprocedural summary from *lp* to *lq* inside a same function f is *valid* if, in the control-flow graph for f, no instruction between *lp* and *lq* is modified or is a call to a modified function.

The correctness of this rule is immediate for intraprocedural summaries (as defined in Section 2.2) since, if the condition stated in the rule holds, we know that all instructions between *lp* and *lq* are unchanged across program versions.

Implementing this rule requires building the control-flow graph of every function containing an old intraprocedural summary and the call graph for the entire program in order to transitively determine which functions are modified. Note that the precision of the rule above could be improved by considering interprocedural control-flow graphs (merging together multiple intraprocedural control-flow graphs), at the cost of building larger graphs.

4 Phase 2: Predicate-Sensitive Change Impact Analysis

Consider the summary $\langle lp, x > 0 \wedge y < 10, lq, w = 0\rangle$ for the code fragment shown on the left of Figure 2. Assume the instructions marked with "MODIFIED" have been modified in the new version. Since some instructions on some paths from *lp* to *lq* have been modified, the Phase 1 analysis will invalidate the summary. However, notice that

```
    ...
lp: if (x > 0) {
      if (y == 10)
        w++; // MODIFIED
      else
        w = 0;
    } else {
        w = 1; // MODIFIED
    }
lq: ...
```

```
    ...
lp: if (x < 0) {
      if (y < 0)
        r = 1;
      else {
        r = 0; //MODIFIED to r = 4;
       }
     }
lq: ...
```

Fig. 2. Motivating examples for Phase 2 (left) and Phase 3 (right)

the set of executions that start from a state satisfying $x > 0 \wedge y < 10$ at *lp* and reach *lq* has not changed.

In this section, we present a second change impact analysis "Phase 2" that exploits the predicates P and Q in a summary $\langle lp, P, lq, Q\rangle$ to perform a more refined analysis. The basic idea is simple: instead of considering all the paths between *lp* and *lq*, we only consider those that also satisfy P in *lp* and Q in *lq*. We now describe how to perform such a predicate-sensitive change impact analysis using static verification-condition generation and theorem proving. We start with a program transformation for checking that all executions satisfying P in *lp* that reach *lq* and satisfy Q in *lq* are not modified from *lp* to *lq*.

Given an intraprocedural summary $\langle lp, P, lq, Q\rangle$ for a function f, we modify the body of f in the *old* code as follows. Let *Entry* denote the location at the beginning of f, i.e., just before the first instruction executed in f. We use an auxiliary Boolean variable modified, and insert the following code at the labels *Entry*, *lp*, *lq* and at all labels ℓ corresponding to a *modified* instruction or a call to a modified function (just before the instruction at that location).

$$\begin{array}{rl} Entry: & \mathsf{goto}\ lp; \\ lp: & \mathsf{assume}\ P;\ \mathsf{modified} := \mathsf{false}; \\ lq: & \mathsf{assert}\ (Q \implies \neg\mathsf{modified}); \\ \ell: & \mathsf{modified} := \mathsf{true}; \end{array}$$

The assume statement assume P at *lp* is a blocking instruction [4], which acts as a no-op if control reaches the statement in a state satisfying the predicate P, and blocks the execution otherwise. The assertion at *lq* checks that if an execution reaches *lq* where it satisfies Q via *lp* where it satisfied P, it does not execute any *modified* instruction between *lp* and *lq*.

Theorem 1. *Given an intraprocedural must summary $\langle lp, P, lq, Q\rangle$ valid for a function f in an old program Prog, if the assertion at lq holds in the instrumented old program for all possible inputs for f, then $\langle lp, P, lq, Q\rangle$ is a valid must summary for the new program Prog′.*

Proof. The assertion at *lq* ensures that *all* executions in the old program *Prog* that (1) reach *lq* and satisfy Q in *lq* and (2) satisfy P at *lp* do not execute any instruction that is marked as *modified* between *lp* and *lq*. This set of executions is possibly over-approximated by considering *all* possible inputs for f, i.e., ignoring specific calling contexts for f and *lp* in *Prog*. Since all the instructions executed from *lp* to *lq* during those executions are preserved in the new program *Prog*′, all those executions W from *lp* to *lq* are still possible in the new program. Moreover, since $\langle lp, P, lq, Q\rangle$ is a must summary for the old program *Prog*, we know that for every state s satisfying P in *lp*, there exists an execution w from s that reaches *lq* and satisfies Q in *lq* in *Prog*. This execution w is included in the set W preserved from *Prog* to *Prog*′. Therefore, by Definition 1, $\langle lp, P, lq, Q\rangle$ is a valid must summary for *Prog*′. □

The reader might wonder the reason for performing the above instrumentation on the old program *Prog* instead of on the new program *Prog*′. Consider the case of a state that satisfies P at *lp* from which there is an execution that reaches *lq* in *Prog*, but from which no execution reaches *lq* in *Prog*′. In this case, the must summary $\langle lp, P, lq, Q\rangle$ is invalid for *Prog*′. Yet applying the above program transformation to *Prog*′ would not necessarily trigger an assertion violation at *lq* since *lq* may no longer be reachable in *Prog*′.

To validate must summaries statically, one can use any static assertion checking tool to check that the assertion in the instrumented program does not fail for all possible function inputs. In this work, we use Boogie [3], a verification condition (VC) based program verifier to check the absence of assertion failures. VC-based program verifiers create a logic formula from a program with assertions with the following guarantee: if the logic formula is valid, then the assertion does not fail in any execution. The validity of the logic formula is checked using a theorem prover, typically a SMT solver. For loop-free and call-free programs, the logic formula is generated by computing variants of *weakest liberal preconditions* (*wlp*) [9]. Procedure calls can be handled by assigning non-deterministic values to the return variable and all the globals that can be potentially modified during the execution of the callee. Similarly, loops can be handled by assigning non-deterministic values to all the variables that can be modified during the execution of the loop. Although procedure postconditions and loop invariants can be used to recover the loss of precision due to the use of non-determinism for over-approximating side effects of function calls and loop iterations, we use the default postcondition and loop invariant true for our analysis to keep the analysis automated and simple.

5 Phase 3: Must Summary Validity Checking

Consider the code fragment shown on the right of Figure 2 where the instruction marked "MODIFIED" is modified in the new code. Consider the summary $\langle lp, x < 0, lq, r \geq 0\rangle$. Since the *modified* instruction is along a path between *lp* and *lq*, even when restricted under the condition P at *lp*, neither Phase 1 nor Phase 2 will validate the summary. However, note that the change does not affect the validity of the must summary: all executions satisfying $x < 0$ at *lp* still reach *lq* and satisfy $r \geq 0$ in the new code, which means the must summary is still valid. In this section, we describe a third algorithm dubbed "Phase 3" for statically checking the validity of a must summary $\langle lp, P, lq, Q\rangle$ against some code, *independently of code changes*.

In the rest of this section, we assume that the programs under consideration are (i) *terminating*, i.e., every execution eventually terminates, and (ii) *complete*, i.e., every state has a successor state.

Given an intraprocedural summary ⟨*lp, P, lq, Q*⟩ for a function f, we perform the following instrumentation on the *new* code. We denote by *Entry* the location of the first instruction in f, while *Exit* denotes any exit instruction in f. We use an auxiliary Boolean variable reach_lq, and insert the following code at the labels *Entry*, *lp*, *lq* and *Exit*.

$$\begin{aligned} \mathit{Entry}:&\ \ \mathsf{reach\_lq} := \mathsf{false};\ \mathsf{goto}\ \mathit{lp};\\ \mathit{lp}:&\ \ \mathsf{assume}\ P;\\ \mathit{lq}:&\ \ \mathsf{assert}\ (Q);\ \mathsf{reach\_lq} := \mathsf{true};\\ \mathit{Exit}:&\ \ \mathsf{assert}\ (\mathsf{reach\_lq}); \end{aligned}$$

The variable reach_lq is set when *lq* is visited in an execution, and initialized to false at the *Entry* node. The assume *P* blocks the executions that do not satisfy *P* at *lp*. The assertion at *lq* checks that if an execution reaches *lq* via *lp*, it satisfies *Q*. Finally, the assertion at *Exit* checks that *all* executions from *lp* have to go through *lq*.

Theorem 2. *Given an intraprocedural must summary ⟨lp, P, lq, Q⟩ for a function f, if the assertions hold in the instrumented program for all possible inputs of f, then ⟨lp, P, lq, Q⟩ is a valid must summary for the program.*

Proof. The assertion at *lq* ensures that *every* execution that reaches *lq* from a state satisfying *P* at *lp*, satisfies *Q*. This set of executions is possibly over-approximated by considering *all* possible inputs for f, i.e., ignoring specific calling contexts for f and *lp*. Since we consider programs that are terminating and complete, the assertion at *Exit* is checked for every execution (except those blocked by assume *P* in *lp* which do not satisfy *P*), and ensures that every execution that satisfies *P* at *lp* visits *lq*. The goto *lp* ensures that *lp* is reached from *Entry*, otherwise the two assertions could vacuously hold if *lp* was not reachable or through restricted calling contexts smaller than *P*. □

The assertions in the instrumented function can be checked using any off-the-shelf assertion checker as described in Section 4. Our implementation uses VC generation and a theorem prover to validate the summaries. Since loops and procedure calls are treated conservatively by assigning non-deterministic values to modified variables, the static validation is also approximate and may sometimes fail to validate valid must summaries.

Note that Phase 3 is *not* an instance of the Phase 2 algorithm when every statement is marked as "modified": Phase 3 checks the *new* program while Phase 2 checks the *old* program (see also the remark after Theorem 1).

Moreover, the precision of Phase 3 is incomparable to the precision of Phase 2 (which refines Phase 1). Both Phase 1 and Phase 2 validate a must summary for the new program *assuming* it was a must summary for the old program, whereas Phase 3 provides an absolute guarantee on the new program. At the start of this section, we presented an example of a valid must summary that can be validated by Phase 3 but not by Phase 2.

Conversely, Phase 3 may fail to validate a summary due to the presence of complex code between *lp* and *lq* and imprecision in static assertion checking, while Phase 1 or Phase 2 may be able to prove that the summary is still valid by detecting that the complex code has not been modified.

6 Dealing with Partial Summaries

In practice, tracking *all* inputs and outputs of large program fragments can be problematic in the presence of large or complex heap-allocated data structures or when dealing with library or operating-system calls with possibly unknown side effects. In those cases, the constraints *P* and *Q* can be approximate, i.e., only *partially defined*: *P* constraints only *some* inputs, while *Q* can capture only *some* outputs (side effects). The must summary is then called *partial*, and may be wrong in some other unknown calling context. Constraints containing partial must summaries may generate test cases that will not cover the expected program paths and branches. Such *divergences* [14] can be detected at runtime by comparing the expected program path with the actual program path being taken. In practice, divergences are often observed in dynamic test generation, and partial summaries can still be useful to limit path explosion, even at the cost of some divergences.

Consider the partial summary $\langle lp, x > 0, lq, ret = 1\rangle$ for the function

```
    int k(int x) {
lp:  if ((x > 0) && (vGlobal > 10)) return 1;
     return 0;
lq: }
```

where the input value stored in the global variable `vGlobal` is not captured in the summary, perhaps because it does not depend on a whole-program input. If the value of `vGlobal` is constant, the constraint `(vGlobal > 10)` is always true and can safely be skipped. Otherwise, the partial summary is imprecise: it may be wrong in some calling contexts.

The validity of partial must summaries could be defined in a weaker manner to reflect the fact that they capture only partial preconditions, for instance as follows:

Definition 3. *A* partial *must summary $\langle lp, P, lq, Q\rangle$ is valid for a program Prog if there exists a predicate R on program variables, such that (i) R does not imply* false*, (ii) the support*[1] *of R is disjoint from the support of P, and (iii) $\langle lp, P \wedge R, lq, Q\rangle$ is a must summary for Prog.*

Since *R* is not false, the conditions (ii) and (iii) cannot be vacuously satisfied. Moreover, since the supports of *P* and *R* are disjoint, *R* does not constrain the variables in *P* yet requires that the partial must summary tracks a subset of the inputs (namely those appearing in *P*) precisely.

In practice, it can be hard and expensive to determine whether a must summary is partial or not. Fortunately, any partial must summary can be soundly validated using the

[1] The support of an expression refers to the variables in the expression.

stronger Definition 1, which is equivalent to setting R to true in Definition 3. Phases 1, 2 and 3 are thus all sound for validating partial must summaries.

Validating partial summaries with Definition 3 or full summaries for non-deterministic programs with Definition 1 could be done more precisely with an assertion checker that can reason about alternating existential and universal quantifiers, which is non-standard. It would be interesting to develop such an assertion checker in future work.

7 Recomputing Invalidated Summaries

All the summaries declared valid by Phase 1, 2 or 3 are mapped to the new code and can be reused. In contrast, all invalid summaries need to be recomputed, for instance using a breadth-first strategy in the graph formed by superposing path constraints.

Consider the graph G whose nodes are all the program locations lp and lq mentioned in the old set of test summaries, and where there is an edge from lp to lq for each summary. Note that, by construction [11], every node lq of a summary matches the node lp of the next summary in the whole-program path constraint, unless lq is the last conditional statement in the path constraint or lp is the first one, which we denote by r for "root". By construction, G is a directed acyclic graph.

Consider any invalid summary $\langle lp, P, lq, Q\rangle$ that is closest to the root r of G. Let $\mathcal{P}$ denote the set of paths from r to lp. By construction with a breadth-first strategy, all summaries along all the paths in $\mathcal{P}$ are still valid for the new program version. To recompute the summary $\langle lp, P, lq, Q\rangle$ for the new program, we call the constraint solver with the formula

$$P \wedge \bigvee_{\phi_i \in \mathcal{P}} \phi_i$$

in order to generate a test to exercise condition P at the program location lp (see Section 2.1). Then, we run this test against the new program version and generate a new summary from lp to wherever it leads to (possibly a new lq and Q). This process can be repeated to recompute all invalidated summaries in a breadth-first manner in G.

8 Experimental Results

We now present preliminary results for validating intraprocedural must summaries generated by our tool SAGE [15] for several benchmarks, with a focus on understanding the relative effectiveness of the different approaches.

8.1 Implementation

We have developed a prototype implementation for analyzing x86 binaries, using two existing tools: the Vulcan [10] library to statically analyze Windows binaries, and the Boogie [3] program verifier. We briefly describe the implementation of the different phases in this section.

Our tool takes as input the old program (DLLs), the set of summaries generated by SAGE for the old program, and the new version of the program. We use Vulcan to

		Functions with Changes								Summaries
Benchmark	Functions	M	% M	IM	% IM	U	% U	IU	% IU	(Intraprocedural)
ANI	6978	703	10%	3130	45%	2340	34%	5174	74%	286
GIF	13897	712	5%	4370	31%	3814	27%	8827	64%	288
JPEG	20357	623	3%	6150	30%	7463	37%	12184	60%	517

Fig. 3. Benchmark characteristics

find differences between the two versions of the program, and propagate them interprocedurally. In this work, we focus on the validation of must summaries that are intraprocedural (SAGE classifies summaries as intraprocedural or not at generation time). Intraprocedural summaries that cannot be validated by Phase 1 are further examined by the more precise Phases 2 and 3. For each of those, we conservatively translate the x86 assembly code of the function containing the summary to a function in the Boogie input language, and use the Boogie verifier (which uses the Z3 SMT solver) to validate the summaries using the Phase 2 or Phase 3 checks. Finally, our tool maps the *lp* and *lq* locations of every validated summary from the old program to the new program.

Unfortunately, Boogie currently does not generate a VC if the function under analysis has an *irreducible* control-flow graph [1], although the theory handles it [3]. A function has an irreducible control-flow graph if there is an unstructured loop with multiple entry points into the loop. Such an unstructured loop can arise from two sources: (i) x86 binaries often contain unstructured `goto` statements, and (ii) we add a `goto` *lp* statement in Phases 2 and 3 that might jump inside a loop. Such irreducible graphs appear in roughly 20% of the summaries considered in this section. To circumvent this implementation issue, we report experimental results in those cases where such loops are unrolled a constant number of times (four times). Although we have manually checked that many of these examples will be provable if we had support for irreducible graphs, we can treat those results to indicate the potential of Phase 2 or Phase 3: if their effectiveness is poor after unrolling, it can only be worse without unrolling.

8.2 Benchmarks

Table 3 describes the benchmarks used for our experiments. We consider three image parsers embedded in Windows: ANI, GIF and JPEG. For each of these, we ran SAGE to generate a sample of summaries. The number of DLLs with summaries for the three benchmarks were 3 for ANI, 4 for GIF, and 8 for JPEG. Then, we arbitrarily picked a newer version of each of these DLLs; these were between one and three years newer than the original DLLs. The column “Functions” in Table 3 denotes the total number of functions present in the original DLLs. The columns marked “M”, ”IM”, ”U” and ”IU” denote the number of functions that are “Modified”, “Indirectly Modified” (i.e., calling a modified function), “Unknown” (i.e., calling a function in an unknown DLL or through a function pointer) and “Indirectly Unknown”, respectively. The table also contains the percentage of such functions over the total number of functions. Finally, the “Summaries” column denotes the number of summaries classified as intraprocedural. For all three benchmarks, most summaries generated by SAGE are intraprocedural.

Benchmark	# Summ	Phase 1			Phase 2			Phase 3			All		
		#	%	time	#	%	time	#	%	time	#	%	time
ANI	286	167	58%	8m (3m)	244	85%	37m	86	30%	42m	256	90%	87m
GIF	288	198	69%	12m (4m)	264	92%	23m	90	31%	35m	274	95%	70m
JPEG	517	317	61%	18m (6m)	487	94%	31m	173	33%	37m	501	97%	86m

Fig. 4. Different phases on all the intraprocedural summaries

Although these benchmarks have a relatively small fraction of modified functions (between 3% – 10%), the fraction of functions that can transitively call into these functions can be fairly large (between 30% – 45%). The impact of unknown functions is even more significant, with most functions being marked U or IU. Note that any call to a M, IM, U or IU function would be marked as *modified* in Phase 1 of our validation algorithm (Section 3). Although we picked two versions of each benchmark separated by more than a year, we expect the most likely usage of our tool to be for program versions separated only by a few weeks.

8.3 Results

The three tables (Fig. 4, Fig. 5 and Fig. 6) report the relative effectiveness of the different phases on the previous benchmarks. Each table contains the number of intraprocedural summaries for each benchmark ("# Summ"), the validation done by each of the phases, and the overall validation. For each phase (and overall), we report the number of summaries validated ("#"), the percentage of the total number of summaries validated ("%") and the time (in minutes) taken for the validation. The time reported for Phase 1 includes the time taken for generating the *modified* instructions interprocedurally, and mapping the old summaries to the new code; the fraction of time spent solely on validating the summaries is shown in parenthesis. The failure to prove a summary valid in Phase 2 or Phase 3 could be the result of a counterexample, timeout (100 seconds per summary), or some internal analysis errors in Boogie.

Figure 4 reports the effect of passing *all* the intraprocedural summaries independently to all the three phases. First, note that the total number of summaries validated is quite significant, between 90% and 97%. Phase 1 can validate between 58%–69% of the summaries, Phase 2 between 85%–94% and Phase 3 between 30%–33%. Since Phase 1 is simpler, it can validate the summaries the fastest among the three approaches. The results also indicate that Phase 2 has the potential to validate significantly more summaries than Phase 1 or Phase 3. After a preliminary analysis of the counterexamples for Phase 3, its imprecision seems often due to the partiality of must summaries (see Section 6): many must summaries do not capture enough constraints on states to enable their validation using Phase 3.

To understand the overlap between the summaries validated by each phase, we report the results of the three phases in a "pipeline" fashion, where the summaries validated by an earlier phase are not considered in the later stages. In all the configurations, Phase 1 was allowed to go first because it generates information required for running Phase 2 and Phase 3, and because it is the most scalable as it does not involve a program verifier.

Benchmark	# Summ	Phase 1			Phase 2			Phase 3			All		
		#	%	time	#	%	time	#	%	time	#	%	time
ANI	286	167	58%	8m	77	27%	29m	12	4%	6m	256	90%	43m
GIF	288	198	69%	12m	73	25%	15m	3	1%	1m	274	95%	28m
JPEG	517	317	61%	18m	179	35%	18m	5	1%	5m	501	97%	41m

Fig. 5. Pipeline with Phase 1, Phase 2 and Phase 3

Benchmark	# Summ	Phase 1			Phase 3			Phase 2			All		
		#	%	time	#	%	time	#	%	time	#	%	time
ANI	286	167	58%	8m	30	10%	12m	59	21%	27m	256	90%	47m
GIF	288	198	69%	12m	25	9%	7m	51	18%	12m	274	95%	31m
JPEG	517	317	61%	18m	52	10%	14m	132	26%	14m	501	97%	46m

Fig. 6. Pipeline with Phase 1, Phase 3, Phase 2

The invalid summaries from Phase 1 are passed either to Phase 2 first (Figure 5) or to Phase 3 first (Figure 6).

The results indicate that the configuration of running Phase 1, followed by Phase 2 and then Phase 3 is the fastest. The overall runtime in Figure 5 is roughly half than the overall runtime in Figure 4. Note that the number of additional summaries validated by Phase 3 beyond Phases 1 and 2 is only 1%–4%.

On average from Figure 5, it takes about (43 min divided by 256 summaries) 10 secs to statically validate one summary for ANI, 6 secs for GIF and 5 secs for JPEG. In contrast, the average time needed by SAGE to dynamically re-compute a summary from scratch is about 10 secs for ANI, 70 secs for GIF and 100 secs for JPEG. Statically validating summaries is thus up to 20 times faster for these benchmarks.

9 Related Work

Compositional *may static* program analysis has been amply discussed in the literature [25]. A compositional analysis always involves some form of summarization. Incremental program analysis is also an old idea [7, 24] that nicely complements compositionality. Any incremental analysis involves the use of some kind of "derivation graph" capturing inference interdependencies between summaries during their computation, such as which lower-level summary was used to infer which higher-level summary. While compositional interprocedural analysis has now become mainstream in industrial-strength static analysis tools (e.g., [19]) which otherwise would not scale to large programs, incremental algorithms are much less widely used in practice. Indeed, those algorithms are more complicated and often not really needed as well-engineered compositional static analysis tools can process millions of lines of code in only hours on standard modern computers.

The purpose of our general line of research is to replicate the success of compositional static program analysis to the testing space. In our context, the summaries we memoize (cache) are symbolic test must summaries [2, 11] which are general input-dependent pre/postconditions of a-priori arbitrary code fragments, and which are represented as logic formulas that are used by an SMT solver to carry out the interprocedural part of the analysis. Because test summaries need to be precise (compared to those produced by standard static analysis) and are generated during an expensive dynamic symbolic execution of large whole programs, incrementality is more appealing for cost-reduction in our context.

The algorithms presented in Sections 3 and 4 have the general flavor of incremental algorithms [24], while the graph formed by superposing path constraints and used to recompute invalidated summaries in Section 7 corresponds to the "derivation graph" used in traditional incremental compositional static-analysis algorithms. However, the details of our algorithms are new due to the specific nature of the type of summaries we consider.

The closest related work in the testing space are probably techniques for *regression test selection* (e.g., see [17]) which typically analyze test coverage data and code changes to determine which tests in a given test suite need to be re-executed to cover newly modified code. The techniques we use in Phase 1 of our algorithm are similar, except we do not record coverage data for each pair *lp* and *lq* as discussed at the beginning of Section 3. There is a rich literature on techniques for static and dynamic *change impact analysis* (see [26] for a summary). Our Phase 1 can be seen as a simple instance of these techniques, aimed at validating a given must summary. Although more sophisticated static-analysis techniques (based on dataflow analysis) have been proposed for change impact analysis, we are not aware of any attempt to use verification-condition generation and automated theorem proving techniques like those used in Phase 2 and Phase 3 for precise checking of the impact of a change. The work on *differential symbolic execution* (DSE) [22] is the closest to our Phase 3 algorithm. Unlike DSE, we do not summarize paths in the new program to compare those with summaries of the old program; instead, we want to avoid recomputing new summaries by reusing old ones as much as possible. Whenever an old summary $\langle lp, P, lq, Q\rangle$ becomes invalid and needs to be recomputed, a data-flow-based impact analysis like the one discussed in [23] could refine the procedure described in Section 7 by identifying which specific program paths from *lp* to *lq* need to be re-executed symbolically. In our experiments, every summary covers one or very few paths (of the old program), and this optimization is not likely to help much.

Must abstractions are program abstractions geared towards finding errors, which dualize may abstractions geared towards proving correctness [13]. Reasoning about must abstractions using logic constraint solvers has been proposed before [6, 13, 16, 18, 20], and are related to Phase 3 in our work.

10 Conclusions

In this work, we formulated the problem of statically validating must summaries to make compositional dynamic test generation more incremental. We described three approaches for validating must summaries, that differ in their strengths and weaknesses. We outlined the subtleties involved in using an off-the-shelf verification-condition-based checker for validating must summaries, and the impact of partial predicates on precision. We presented a preliminary evaluation of these approaches on a set of representative intraprocedural summaries generated from real-world applications, and demonstrated the effectiveness of static must summary checking. We plan to evaluate our tool on a larger set of summaries and benchmarks, investigate how to validate interprocedural summaries, and improve the precision of the path-sensitive analysis.

Acknowledgements. We thank the anonymous reviewers for their constructive comments. The work of Cindy Rubio-González was done mostly while visiting Microsoft Research. A preliminary version of this work appeared under the title "Incremental Compositional Dynamic Test Generation" as MSR Technical Report MSR-TR-2010-11, February 2010.

References

1. Aho, A., Sethi, R., Ullman, J.: Compilers: Principles, Techniques and Tools. Addison-Wesley, Reading (1986)
2. Anand, S., Godefroid, P., Tillmann, N.: Demand-Driven Compositional Symbolic Execution. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 367–381. Springer, Heidelberg (2008)
3. Barnett, M., Chang, B.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2005)
4. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In: PASTE 2005, pp. 82–87 (2005)
5. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: Automatically Generating Inputs of Death. In: ACM CCS (2006)
6. Chandra, S., Fink, S.J., Sridharan, M.: Snugglebug: A Powerful Approach to Weakest Preconditions. In: PLDI 2009 (2009)
7. Conway, C.L., Namjoshi, K.S., Dams, D., Edwards, S.A.: Incremental algorithms for interprocedural analysis of safety properties. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 449–461. Springer, Heidelberg (2005)
8. de Moura, L., Bjorner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
9. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM 18, 453–457 (1975)
10. Edwards, A., Srivastava, A., Vo, H.: Vulcan: Binary transformation in a distributed environment. Technical report, MSR-TR-2001-50, Microsoft Research (2001)
11. Godefroid, P.: Compositional Dynamic Test Generation. In: POPL 2007, pp. 47–54 (2007)
12. Godefroid, P.: Software Model Checking Improving Security of a Billion Computers. In: Păsăreanu, C.S. (ed.) Model Checking Software. LNCS, vol. 5578, pp. 1–1. Springer, Heidelberg (2009)
13. Godefroid, P., Huth, M., Jagadeesan, R.: Abstraction-Based Model Checking Using Modal Transition Systems. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 426–440. Springer, Heidelberg (2001)
14. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed Automated Random Testing. In: PLDI 2005, pp. 213–223 (2005)
15. Godefroid, P., Levin, M., Molnar, D.: Automated Whitebox Fuzz Testing. In: NDSS 2008, pp. 151–166 (2008)

16. Godefroid, P., Nori, A., Rajamani, S., Tetali, S.: Compositional Must Program Analysis: Unleashing The Power of Alternation. In: POPL 2010 (2010)
17. Graves, T.L., Harrold, M.J., Kim, J.-M., Porter, A., Rothermel, G.: An Empirical Study of Regression Test Selection Techniques. ACM Transactions on Software Engineering and Methodology (TOSEM) 10(2), 184–208 (2001)
18. Gurfinkel, A., Wei, O., Chechik, M.: Yasm: A Software Model-Checker for Verification and Refutation. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 170–174. Springer, Heidelberg (2006)
19. Hallem, S., Chelf, B., Xie, Y., Engler, D.: A System and Language for Building System-Specific Static Analyses. In: PLDI 2002, pp. 69–82 (2002)
20. Hoenicke, J., Leino, K.R.M., Podelski, A., Schäf, M., Wies, T.: It's doomed; we can prove it. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 338–353. Springer, Heidelberg (2009)
21. Molnar, D., Li, X.C., Wagner, D.: Dynamic test generation to find integer bugs in x86 binary linux programs. In: Proc. of the 18th Usenix Security Symposium (2009)
22. Person, S., Dwyer, M.B., Elbaum, S.G., Pasareanu, C.S.: Differential symbolic execution. In: SIGSOFT FSE, pp. 226–237 (2008)
23. Person, S., Yang, G., Rungta, N., Khurshid, S.: Directed Incremental Symbolic Execution. In: PLDI 2011, pp. 504–515 (2011)
24. Ramalingam, G., Reps, T.: A Categorized Bibliography on Incremental Algorithms. In: POPL 1993, pp. 502–510 (1993)
25. Reps, T., Horwitz, S., Sagiv, M.: Precise Interprocedural Dataflow Analysis via Graph Reachability. In: POPL 1995, pp. 49–61 (1995)
26. Santelices, R.A., Harrold, M.J., Orso, A.: Precisely detecting runtime change interactions for evolving software. In: ICST, pp. 429–438 (2010)
27. Satisfiability Modulo Theories Library (SMT-LIB), http://goedel.cs.uiowa.edu/smtlib/

On Sequentializing Concurrent Programs*

Ahmed Bouajjani[1], Michael Emmi[1,**], and Gennaro Parlato[2]

[1] LIAFA, Université Paris Diderot, France
{abou,mje}@liafa.jussieu.fr
[2] School of Electronics and Computer Science
University of Southampton, UK
gennaro@ecs.soton.ac.uk

Abstract. We propose a general framework for compositional underapproximate concurrent program analyses by reduction to sequential program analyses—so-called *sequentializations*. We notice the existing sequentializations—based on bounding the number of execution contexts, execution rounds, or delays from a deterministic task-schedule—rely on three key features for scalable concurrent program analyses: (i) reduction to the *sequential* program model, (ii) *compositional* reasoning to avoid expensive task-product constructions, and (iii) *parameterized* exploration bounds. To understand how those sequentializations can be unified and generalized, we define a general framework which preserves their key features, and in which those sequentializations are particular instances. We also identify a most general instance which considers more executions, by composing the rounds of different tasks in any order, restricted only by the unavoidable program and task-creation causality orders. In fact, we show this general instance is fundamentally more powerful by identifying an infinite family of state-reachability problems (to states $g_1, g_2, \ldots$) which can be answered precisely with a fixed exploration bound, whereas the existing sequentializations require an increasing bound k to reach each g_k. Our framework applies to a general class of shared-memory concurrent programs, with dynamic task-creation and arbitrary preemption.

1 Introduction

Concurrent program analysis is difficult due to the high computational complexity that arises when considering every intricate task interleaving. To avoid such high computational complexity, bounded (underapproximating) exploration is emerging as a general technique. In general, one characterizes a subset of the concurrent program semantics by a bounding parameter k. As k is increased we explore more behaviors, at the expense of more computational resources; in the limit we explore every behavior. A bounded exploration technique is effective when useful information (e.g., the presence of bugs) is obtained by spending a relatively small amount of computational resources (i.e., using low values of k).

[*] Partially supported by the project ANR-09-SEGI-016 Veridyc.

[**] Supported by a fellowship from the Fondation Sciences Mathématiques de Paris.

E. Yahav (Ed.): SAS 2011, LNCS 6887, pp. 129–145, 2011.

By characterizing a subset of the concurrent program semantics by a bounding parameter (e.g., bounded context-switch [22]), it is often possible to reduce concurrent program exploration to sequential program exploration [23, 20, 15, 17, 8, 14]—essentially by constructing an equivalent program without concurrency primitives (e.g., task creation, preemption). Such *sequentializations* are desirable, both practically and theoretically, for several reasons. First, they reduce the exploration problem of any concurrent program to the well-understood model of sequential programs, and are not limited by finite-data programs, per se. Here several practical verification algorithms exist by way of finite-data software model checking [24, 5, 3, 16]), (infinite-state) fixed point computations [6, 25, 12], and (e.g., SMT-based) bounded software verification algorithms [7, 18]. Second, they compute the behavior of each concurrent task compositionally—i.e., without considering the local-state valuations of other tasks. Third, they enable incremental analysis, trading-off computing resources for the amount of explored behaviors, by varying a bounding parameter $k \in \mathbb{N}$. (The parameter k determines the (asymptotic) budget of the sequential program exploration. In practice, the reductions add $\mathcal{O}(k)$ variables to the resulting sequential program, resulting in an increasingly-expensive (in k) sequential program exploration.) Indeed these sequentialization-based analyses have been applied to discover (in some cases previously-unknown) bugs in device drivers [20, 15, 17, 19, 8].

Our principal aim in this work is to develop a theory of parameterized, compositional, concurrent-to-sequential program analysis reductions. Specifically, we make the following contributions:

- To identify the fundamental mechanisms *enabling* compositional sequentializations, thus formulating a framework in which to define them (Section 3).
- To formulate a most general sequentialization in our framework which expresses as many concurrent behaviors as possible (Section 4), while maintaining compositionality, and without extending the class of programs in which tasks are originally written—e.g., by adding unbounded counters.
- To classify the existing sequentializations in our framework, and compare them w.r.t. the behaviors explored for their bounding parameters (Section 5).

Besides enlightening the mechanisms which enable sequential reduction, we believe our gained insight can guide further advances, for instance by considering other restrictions to our framework with efficient encodings.

Using the three features discussed above ((i) reduction to sequential programs, (ii) compositionality, and (iii) parameterization), the existing sequentializations work by characterizing each concurrent task by an interface of k global-state valuation pairs (where k is the bounding parameter). These k global-state valuation pairs represent a computation of a task which is interrupted $k-1$ times—the valuations give the initial and final global-states per execution "round." Each task's interface is then computed—in isolation from other tasks—by beginning each round from the guessed global-state valuation, performing a sequence of sequential program steps, then eventually moving to the next round, by updating the current global-state to the next-guessed valuation. Contiguous interfaces (i.e., where the final global-state valuations of one matches the initial valuations of the other) are then glued together to form larger computations. Intuitively, this interface composition builds executions according to a round-robin schedule of k rounds; a complete execution is formed when each i^{th} final global-state of the last task's interface matches the $(i+1)^{\text{st}}$ initial global-state of the first's. When there are a fixed number of statically-created tasks, interfaces are simply composed in task-identifier order. In the more general case, with *dynamically* created tasks, interfaces are composed in a depth-first preorder over the ordered task-creation tree. (Note by viewing the task-identifier order as the task-creation order, the dynamic case subsumes the static.)

Thus, besides features/constraints (i), (ii), and (iii)—which we feel are desirable, and important to maintain for efficiency and scalability of the resulting program analyses—the existing sequentializations are constrained by: (iv) round-robin executions (in depth-first order on the ordered task-creation tree). Though other schedules can be simulated by increasing the (round) bounding parameter k when the number of tasks is bounded, for a fixed value of k, only k-round round-robin schedules are explored.

The most general instance of the framework we propose in Section 4 subsumes and extends the existing round-robin based (i.e., context- and delay-bounded) sequentializations. As in these previous approaches, we restrict ourselves to explorations that are (i) sequential, (ii) compositional, and (iii) parameterized, in order to compute task interfaces over k global-state valuation pairs. However, for the same analysis budget[1] (i.e., the bounding parameter k), the general instance expresses many more concurrent behaviors, essentially by relaxing the round-robin execution order, and decoupling each task from any global notion of "rounds." We consider that each task's interface is constructed by composing the rounds of tasks it has created in *any* order that respects program order, and task-creation causality order—i.e., a task is not allowed to execute before a subsequent preemption of the task that created it. Thus the simulated concurrent execution need not follow a round-robin schedule, despite the prescribed task traversal order; the possible inter-task interleavings are restricted only by the constraint that at most k sub-task round summaries can be kept at any moment—indeed this constraint is necessary to encode the resulting sequentialization with a bounded number of additional (finite-domain) program variables.

In fact, the most general instance of our framework is fundamentally more expressive than the existing sequentializations, since there are infinite sequences of states (e.g., $g_1, g_2, \ldots$) which can be reached with a fixed exploration bound, whereas the existing sequentializations require an increasing bound k to reach each g_k. This gain in expressiveness may or may not be accompanied by an analysis overhead. For enumerative program analysis techniques (e.g., model checking, systematic testing), one may argue that considering more interleavings will have a negative impact on scalability. This argument is not so clear, however, for symbolic techniques (e.g., SMT-based verification, infinite-state fixed-point computations), which may be able to reduce reasoning over very many interleavings to very few, in practice [20]. Although many device driver bugs have been

[1] See Section 4 for complexity considerations.

discovered within few interleavings [21], reducing the number of interleavings could indeed cause missed bugs in other settings. Furthermore, our hope is that enlightening the mechanisms behind sequentialization will lead to the discovery of other succinctly-encodable instances of compositional sequentialization.

2 Concurrent Programs

We consider a simple but general concurrent programming model in the style of single-threaded event-driven programs. (The style is typically used as a lightweight technique for adding reactivity to single-threaded applications by breaking up long-running computations into a collection of *tasks*.[2]) In this model, control begins with an initial task, which is essentially a sequential program that can read from and write to global (i.e., shared) storage, and *post* additional tasks (each **post** statement specifies a procedure name and argument) to an initially empty *task buffer* of pending tasks. When control returns to the dispatcher, either when a task *yields* control—in which case it is put back into the task buffer—or completes its execution, the dispatcher picks some task from the task buffer, and transfers control to it; when there are no pending tasks to pick, the program terminates. This programming model is powerful enough to model concurrent programs with arbitrary preemption and synchronization, e.g., by inserting a **yield** statement before every shared variable access.

Let Procs be a set of procedure names, Vals a set of values containing `true` and `false`, and T the sole type of values. The grammar of Fig. 1 describes our language of *asynchronous programs*, where p ranges over procedure names. We intentionally leave the syntax of expressions e unspecified, though we do insist the set of expressions Exprs contains Vals and the *(nullary) choice operator* $\star$. The set of program statements s is denoted Stmts. A *sequential program* is an asynchronous program which contains neither **post** nor **yield** statements.

A *(procedure-stack) frame* $\langle \ell, s \rangle$ is a valuation $\ell \in \mathsf{Vals}$ to the procedure-local variable `l`, along with a statement s to be executed. (Here s describes the entire body of a procedure p that remains to be executed, and is initially set to p's top-level statement s_p.) A *task* w is a sequence of frames representing a procedure stack, and the set $(\mathsf{Vals} \times \mathsf{Stmts})^*$ of tasks is denoted Tasks. An *(asynchronous) configuration* $c = \langle g, w, m \rangle$ is a global-variable valuation $g \in \mathsf{Vals}$ along with a task $w \in \mathsf{Tasks}$, and a task buffer multiset $m \in \mathbb{M}[\mathsf{Tasks}]$.

To define the transition relation between configurations, we assume the existence of an evaluation function $[\![\cdot]\!]_{\mathrm{e}} : \mathsf{Exprs} \to \wp(\mathsf{Vals})$ for expressions without program variables, such that $[\![\star]\!]_{\mathrm{e}} = \mathsf{Vals}$. For convenience, we define $e(g, \ell) \overset{\text{def}}{=} [\![e[g/\mathtt{g}, \ell/\mathtt{l}]]\!]_{\mathrm{e}}$ —since `g` and `l` are the only variables, the expression $e[g/\mathtt{g}, \ell/\mathtt{l}]$ has no free variables. To select the next-scheduled statement in a configuration, we define a *statement context* S as a term derived from the grammar $S ::= \diamond \mid S;\ s$, and write $S[s]$ for the statement obtained by substituting a statement s for the unique occurrence of $\diamond$ in S.

[2] As our development is independent from architectural particularities, "tasks" may correspond to threads, processes, asynchronous methods, etc.

$$
\begin{array}{rcl}
P & ::= & \textbf{var}\ \texttt{g}{:}T\ \ H^{*} \\
H & ::= & \textbf{proc}\ p\ \texttt{(}\textbf{var}\ \texttt{l}{:}T\texttt{)}\ s \\
s & ::= & s\texttt{;}\ s \quad | \quad x\ \texttt{:=}\ e \quad | \quad \textbf{skip} \quad | \quad \textbf{assume}\ e \\
 & | & \textbf{if}\ e\ \textbf{then}\ s\ \textbf{else}\ s \quad | \quad \textbf{while}\ e\ \textbf{do}\ s \\
 & | & \textbf{call}\ x\ \texttt{:=}\ p\ e \quad | \quad \textbf{return}\ e \\
 & | & \textbf{post}\ p\ e \quad | \quad \textbf{yield} \\
x & ::= & \texttt{g} \quad | \quad \texttt{l}
\end{array}
$$

Fig. 1. The grammar of asynchronous programs. Each program P declares a single type-T global variable $\texttt{g}$, and a sequence of procedures named $p_1 \ldots p_n \in \mathsf{Procs}^*$. Each procedure p has single type-T parameter $\texttt{l}$, and a top-level statement (i.e., the procedure body) denoted s_p. This program syntax is made simple only to simplify presentation; various extensions—e.g., to multiple global and local variables—can be encoded by varying the type T; see the full version of this paper [4].

$$
\textsc{Post}\quad \frac{\ell' \in e(g,\ell) \qquad w' = \langle \ell', s_p \rangle}{\langle g, \langle \ell, S[\textbf{post}\ p\ e] \rangle\, w, m \rangle \rightarrow^{\mathrm{a}}_{P} \langle g, \langle \ell, S[\textbf{skip}] \rangle\, w, m \cup \{w'\} \rangle}
\qquad
\textsc{Dispatch}\quad \frac{w \in m}{\langle g, \varepsilon, m \rangle \rightarrow^{\mathrm{a}}_{P} \langle g, w, m \setminus \{w\} \rangle}
$$

$$
\textsc{Complete}\quad \frac{}{\langle g, \langle \ell, S[\textbf{return}\ e] \rangle, m \rangle \rightarrow^{\mathrm{a}}_{P} \langle g, \varepsilon, m \rangle}
\qquad
\textsc{Yield}\quad \frac{w' = \langle \ell, S[\textbf{skip}] \rangle\, w}{\langle g, \langle \ell, S[\textbf{yield}] \rangle\, w, m \rangle \rightarrow^{\mathrm{a}}_{P} \langle g, \varepsilon, m \cup \{w'\} \rangle}
$$

Fig. 2. The transition relation $\rightarrow^{\mathrm{a}}_{P}$ for an asynchronous program P is given by combining the transitions above with the sequential transition relation $\rightarrow^{\mathrm{s}}_{P}$.

The transition relation $\rightarrow^{\mathrm{a}}_{P}$ of an asynchronous program P is defined in Fig. 2 as a set of operational steps on configurations. The transition relation $\rightarrow^{\mathrm{s}}_{P}$ for the sequential program statements (see the full version of this paper [4]) is standard. The Post rule gives a newly-created task to the task buffer, while the Yield rule gives the currently-executing task to the task buffer. The Dispatch rule chooses some task from the buffer to execute, and the Complete rule disposes a completed task.

A configuration $\langle g, \langle \ell, s \rangle, \emptyset \rangle$ is called *initial*, and is *sequential* when s does not contain **post** (nor **yield**) statements. An *(asynchronous) execution of a program P (from c_0 to c_j)* is a configuration sequence $h = c_0 c_1 \ldots c_j$ where

- c_0 is initial, and
- $c_i \rightarrow^{\mathrm{a}}_{P} c_{i+1}$ for $0 \leq i < j$.

We say a configuration $c = \langle g, w, m \rangle$ (alternatively, the global value g) is *reachable in P (from c_0)* when there exists an execution of P from c_0 to c. The *asynchronous semantics* of P, written $\{\!|P|\!\}_{\mathrm{a}}$, maps initial configurations to reachable global values, i.e., $\{\!|P|\!\}_{\mathrm{a}}(c_0) = g$ if and only if g is reachable in P from c_0. When P is a sequential program, the *sequential semantics* of P, written $\{\!|P|\!\}_{\mathrm{s}}$, maps initial sequential configurations to reachable global values.

3 Compositional Semantics

Here we define a *compositional* semantics for asynchronous programs on which to base our reduction to sequential programs. To do so, we characterize each posted task by an *interface* exposing only global-state valuations to other tasks. Each interface summarizes not only the computation of a single task, but also the computations of all descendants of tasks it has posted. In particular, an interface is a sequence $\langle g_1, g_1' \rangle \ldots \langle g_k, g_k' \rangle$ of global-state valuation pairs summarizing a computation which is interrupted $k-1$ times; the computation begins with global state g_1, is interrupted by an external task at global state g_1', is resumed at g_2, etc. We call the computation summarized by each pair $\langle g_i, g_i' \rangle$ *the* i^{th} *round*. A larger computation is then formed from a collection of task interfaces, by gluing together contiguous rounds (e.g., summarized by $\langle g_1, g_2 \rangle$ and $\langle g_2, g_3 \rangle$), while maintaining the order between rounds from the same interface.

We construct interfaces inside a data-structure called a *summary bag*. Besides the sequence $\mathcal{B}_{\mathrm{ex}}$ of rounds which will be exported as the current task's interface (see Fig. 3a), the bag maintains a collection $\mathcal{B}_{\mathrm{im}}$ of interfaces imported from posted tasks (see Fig. 3c). At any yield-point, the current task can begin a new round, by *guessing*[3] the global-state valuation that will begin the next round (see Fig. 3b), or, if the current global-state valuation matches the start of the first unconsumed round from some imported interface (as does $\langle a, b \rangle$ in Fig. 3d), the current task can consume that round and update the current global state; this amounts to interleaving the round of a posted task at the current control point.

With this view of recursive interface construction—interfaces are constructed from interfaces of sub-tasks—the reduction to sequential programs is nearly straightforward. To compute the imported summary of a posted task, we translate the **post** statement into a procedure call which returns the computed interface for the posted task. At yield points, we will repeatedly, nondeterministically choose to begin a new round, consume a round from an imported interface, or step past the **yield** statement. For the moment the only obstacle is how to store the unbounded summary bag; we address this issue in Section 4.

To define the compositional semantics we formalize the summary-bag operations. A *summary bag* $\mathcal{B} = \langle \mathcal{B}_{\mathrm{ex}}, \mathcal{B}_{\mathrm{im}} \rangle$ pairs the round sequence $\mathcal{B}_{\mathrm{ex}}$ to be exported from the current task, along with a collection $\mathcal{B}_{\mathrm{im}}$ of round sequences imported from sub-tasks. The *empty bag* $\mathcal{B}_\emptyset = \langle \varepsilon, \emptyset \rangle$ is an empty sequence paired with an empty collection. The operations to add the current round ($\oplus$), import a sub-task interface ($\odot$), and consume a sub-task round ($\ominus$) are defined as

$$\begin{aligned} \langle \mathcal{B}_{\mathrm{ex}}, \mathcal{B}_{\mathrm{im}} \rangle \oplus \langle g, g' \rangle &\stackrel{\text{def}}{=} \langle \mathcal{B}_{\mathrm{ex}} \cdot \langle g, g' \rangle, \mathcal{B}_{\mathrm{im}} \rangle, && \text{add round} \\ \langle \mathcal{B}_{\mathrm{ex}}, \mathcal{B}_{\mathrm{im}} \rangle \odot {\mathcal{B}_{\mathrm{ex}}}' &\stackrel{\text{def}}{=} \left\langle \mathcal{B}_{\mathrm{ex}}, \mathcal{B}_{\mathrm{im}} \cup {\mathcal{B}_{\mathrm{ex}}}' \right\rangle, && \text{import interface} \\ \langle \mathcal{B}_{\mathrm{ex}}, \mathcal{B}_{\mathrm{im}} \rangle \ominus \langle g, g' \rangle &\stackrel{\text{def}}{=} \left\langle \mathcal{B}_{\mathrm{ex}}, {\mathcal{B}_{\mathrm{im}}}' \right\rangle && \text{consume round,} \end{aligned}$$

[3] An enumerative analysis algorithm can handle "guessing" by attempting every reached global valuation; a symbolic algorithm just creates a new symbol.

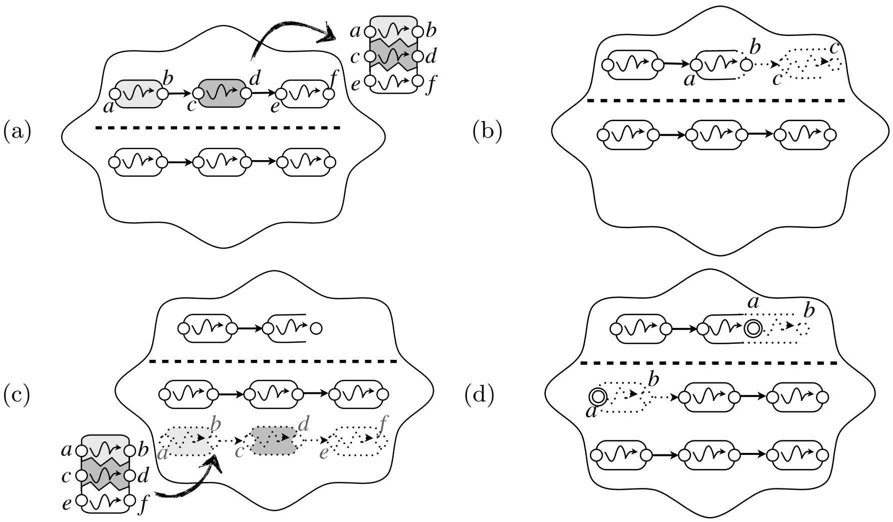


Fig. 3. The summary bag operations. Circles represent global valuations (double circles equivalences), and are paired together in rounded rectangle round summaries. Interfaces are drawn as stacked round summaries, and shading is used only to easily identify summaries. (a) Export a constructed interface, (b) begin a round by finalizing the current round $\langle a, b\rangle$, and guessing a valuation c to begin the next, (c) import sub-task interface, (d) interleave the first unconsumed round $\langle a, b\rangle$ of a sub-task, updating the current global state from a to b. The round-summaries $\mathcal{B}_{\text{ex}}$ to be exported appear above the dotted line, and the collection $\mathcal{B}_{\text{im}}$ of imported sub-task interfaces appears below.

where $\mathcal{B}_{\text{im}}{}'$ is obtained by removing $\langle g, g'\rangle$ from the head of a sequence in $\mathcal{B}_{\text{im}}$, and $\langle \mathcal{B}_{\text{ex}}, \mathcal{B}_{\text{im}}\rangle \ominus \langle g, g'\rangle$ is undefined if $\langle g, g'\rangle$ is not at the head of $\mathcal{B}_{\text{im}}$.

We define the compositional transition relation of asynchronous programs by augmenting the sequential transitions $\rightarrow_P^{\text{s}}$ with a new set of transitions for the asynchronous control statements, over a slightly different notion of configuration. In order to build interfaces, each configuration carries the global-state valuation at the beginning of the current round (denoted g_0 below), as well as the current valuation. A *(compositional) configuration* $c = \langle g_0, g, w, \mathcal{B}\rangle$ is an *initial* valuation $g_0 \in \mathsf{Vals}$ of the global variable g, along with a *current* valuation $g \in \mathsf{Vals}$, a task $w \in \mathsf{Tasks}$, and a summary bag $\mathcal{B}$.

Fig. 4 lists the transitions $\rightarrow_P^{\text{c}}$ for the asynchronous control statements. The NextRound rule begins a new round with a guessed global-state valuation, and the SubTask rule collects the posted task's interface. The task-summarization relation $p\ \ell' \rightsquigarrow \mathcal{B}_{\text{ex}}$ used in the SubTask rule holds when the task $\langle \ell', s_p\rangle$ can execute to a yield, or return, point having constructed the interface $\mathcal{B}_{\text{ex}}$—i.e., when there exists $g, g_0, g_1 \in \mathsf{Vals}$, $w \in \mathsf{Tasks}$, and $\mathcal{B}_{\text{im}}$ such that $\langle g, g, \langle \ell', s_p\rangle, \mathcal{B}_\emptyset\rangle \rightarrow_P^{\text{c}*} \langle g_0, g_1, w, \langle \mathcal{B}_{\text{ex}}, \mathcal{B}_{\text{im}}\rangle\rangle$, where w is of the form $\langle \ell, S[\textbf{yield}]\rangle\, w'$ or $\langle \ell, S[\textbf{return}\ e]\rangle$. The Interleave rule executes a round imported from some posted task, and finally, the Resume rule simply steps past the **yield** statement.

NextRound

$$\frac{\mathcal{B}' = \mathcal{B} \oplus \langle g_0, g \rangle \qquad g' \in \mathsf{Vals}}{\langle g_0, g, \langle \ell, S[\mathbf{yield}] \rangle\, w, \mathcal{B} \rangle \to^{\mathrm{c}}_{P} \langle g', g', \langle \ell, S[\mathbf{yield}] \rangle\, w, \mathcal{B}' \rangle}$$

SubTask

$$\frac{\ell' \in e(g, \ell) \qquad p\ \ell' \rightsquigarrow \mathcal{B}_{\mathrm{ex}} \qquad \mathcal{B}' = \mathcal{B} \odot \mathcal{B}_{\mathrm{ex}}}{\langle g_0, g, \langle \ell, S[\mathbf{post}\ p\ e] \rangle\, w, \mathcal{B} \rangle \to^{\mathrm{c}}_{P} \langle g_0, g, \langle \ell, S[\mathbf{skip}] \rangle\, w, \mathcal{B}' \rangle}$$

Interleave

$$\frac{\mathcal{B}' = \mathcal{B} \ominus \langle g, g' \rangle}{\langle g_0, g, \langle \ell, S[\mathbf{yield}] \rangle\, w, \mathcal{B} \rangle \to^{\mathrm{c}}_{P} \langle g_0, g', \langle \ell, S[\mathbf{yield}] \rangle\, w, \mathcal{B}' \rangle}$$

Resume

$$\frac{}{\langle g_0, g, \langle \ell, S[\mathbf{yield}] \rangle\, w, \mathcal{B} \rangle \to^{\mathrm{c}}_{P} \langle g_0, g, \langle \ell, S[\mathbf{skip}] \rangle\, w, \mathcal{B} \rangle}$$

Fig. 4. The compositional transition relation $\to^{\mathrm{c}}_{P}$ for an asynchronous program P is given by combining the transitions above with the sequential transition relation $\to^{\mathrm{s}}_{P}$

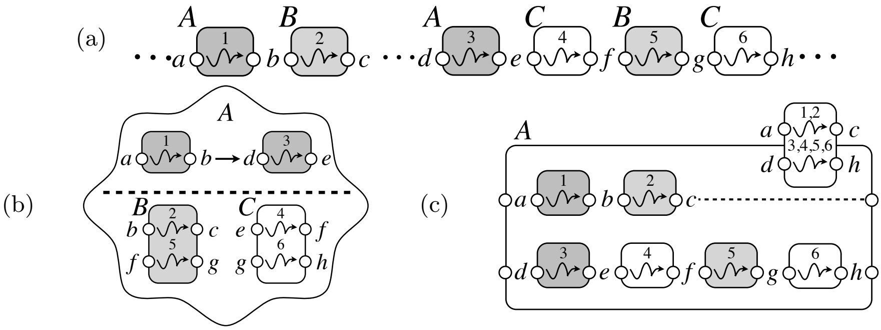


Fig. 5. Simulating an asynchronous execution (a) as a compositional execution, where task A posts B and C, then is interrupted by B, then is eventually re-dispatched, and upon completion is followed directly by C, which is interrupted by B before completing. (b) The bag used to reconstruct A's interface, and (c) the constructed interface for A.

A configuration $\langle g, g, \langle \ell, s \rangle, \mathcal{B}_{\emptyset} \rangle$ is called *initial.* A *(compositional) execution of a program P (from c_0 to c_j)* is a configuration sequence $h = c_0 c_1 \ldots c_j$ where

- c_0 is initial, and
- $c_i \to^{\mathrm{c}}_{P} c_{i+1}$ for $0 \le i < j$.

A compositional execution describes the progression of one task only; progressions of sub-tasks are considered recursively as separate executions to compute the task-summarization relation $\rightsquigarrow$. We say a configuration $c = \langle g_0, g, w, \mathcal{B} \rangle$ (alternatively, the global value g) is *reachable in P (from c_0)* when there exists an execution from c_0 to c, without using the NextRound rule.[4] The *compositional semantics* of P, written $\{\!|P|\!\}_{\mathrm{c}}$, maps initial configurations to reachable global values, i.e., $\{\!|P|\!\}_{\mathrm{c}}(c_0) = g$ if and only if g is reachable in P from c_0.

Although their definitions are wildly different, the compositional and asynchronous semantics are equivalent. To see that every asynchronous execution h has a corresponding compositional execution, consider, inductively, how to build the interface summarizing the projection of h a given task's sub-task segments. Consider the task A of Fig. 5 which posts B and C. To simulate the asynchronous

[4] Disallowing use of the NextRound rule in the top-level execution ensures that unchecked guessed global-state valuations are not considered reachable.

(sub-) execution of Fig. 5a, A builds an interface with two uninterruptible rounds (see Fig. 5c): the first sequencing segments 1 and 2, and the second sequencing 3, 4, 5, and 6. To build this interface, A must import two-round interfaces from B and C each; note that each round of B and C may recursively contain many interleaved segments of sub-tasks.

Theorem 1. *The compositional semantics and asynchronous semantics are identical, i.e., for all programs P we have $\{\!|P|\!\}_{\mathrm{c}} = \{\!|P|\!\}_{\mathrm{a}}$.*[5]

The proof to this theorem appears in the full version of this paper [4].

4 Bounded Semantics

As earlier sequentializations have done by constraining the number of rounds [20, 15], by constraining the size of the summary bag, we can encode the bag contents with a fixed number of additional variables in the resulting sequential program. Since we are interested in exploring as many behaviors as possible with a limited-size bag, an obvious point of attention is *bag compression*. Thus we define an additional operation to free a space in the bag by merging two contiguous (w.r.t. the global valuation) summaries, essentially removing the possibility for future reordering between the selected segments. In doing so, we must maintain causal order by ensuring the bag remains acyclic, and what was before a collection $\mathcal{B}_{\mathrm{im}}$ of summary sequences imported from sub-tasks now becomes a directed acyclic graph (DAG) of summaries. To maintain program order in the presence of merging, summaries are now consumed only from the roots of $\mathcal{B}_{\mathrm{im}}$.

The *size* $|\mathcal{B}| \overset{\mathrm{def}}{=} |\mathcal{B}_{\mathrm{ex}}| + |\mathcal{B}_{\mathrm{im}}|$ of $\mathcal{B}$ is the sum of the length of the sequence $\mathcal{B}_{\mathrm{ex}}$ and the number of nodes of the DAG $\mathcal{B}_{\mathrm{im}}$. The bag simplification operation $\langle \mathcal{B}_{\mathrm{ex}}, \mathcal{B}_{\mathrm{im}} \rangle \rhd \langle \mathcal{B}_{\mathrm{ex}}, {\mathcal{B}_{\mathrm{im}}}' \rangle$ obtains ${\mathcal{B}_{\mathrm{im}}}'$ from $\mathcal{B}_{\mathrm{im}}$ by merging two nodes n and n', labelled, resp., by $\langle g_1, g_2 \rangle$ and $\langle g_2, g_3 \rangle$, such that either

(a) there is no directed path from n to n' in $\mathcal{B}_{\mathrm{im}}$, or
(b) there is an edge from n to n' in $\mathcal{B}_{\mathrm{im}}$

(see Fig. 6); in either case the merged node is labelled $\langle g_1, g_3 \rangle$. Note that when $\mathcal{B} \rhd \mathcal{B}'$ we have $|\mathcal{B}'| = |\mathcal{B}| - 1$. Though we could simulate this merge operation in the (unbounded) compositional semantics, by eventually consuming consecutively n and n' into the exported summary list, the merge operation allows to eagerly express the interleaving—though disallows any subsequent interleaving between n and n'. Merging summaries eagerly fixes a sequence of inter-task execution segments, trading the freedom for external interleaving (which may have uncovered additional, potentially buggy, behaviors) for an extra space in the bag.

We define the *k-bounded compositional semantics of P*, written $\{\!|P|\!\}_{\mathrm{c}}^{k}$, by restricting the compositional semantics $\{\!|P|\!\}_{\mathrm{c}}$ to executions containing only configurations $\langle g_0, g, w, \mathcal{B} \rangle$ such that $|\mathcal{B}| \leq k$, and adding the additional transition

[5] We consider initial configurations $\langle g, \langle \ell, s \rangle, \emptyset \rangle$ and $\langle g, g, \langle \ell, s \rangle, \mathcal{B}_{\emptyset} \rangle$ as equal.

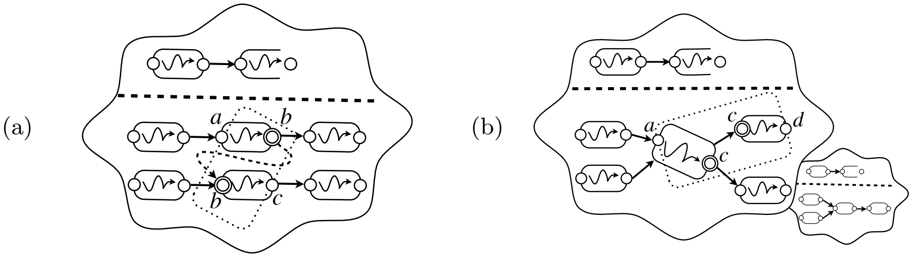


Fig. 6. The bag simplification operations: (a) merging two disconnected but contiguous round summaries $\langle a, b\rangle$ and $\langle b, c\rangle$ (resulting in (b)), and (b) merging two consecutive and contiguous summaries $\langle a, c\rangle$ and $\langle c, d\rangle$ (resulting in the small adjacent bag).

$$\textsc{Compress}\quad \frac{\mathcal{B} \rhd \mathcal{B}'}{\langle g_0, g, w, \mathcal{B}\rangle \to^{\mathrm{c}}_{P} \langle g_0, g, w, \mathcal{B}'\rangle}.$$

As we increase k, the set of k-bounded semantics grows monotonically, and in the limit, the k-bounded semantics *is* the compositional semantics. For two functions $f, g : A \to \wp(B)$, we write $f \subseteq g$ when for all $a \in A$, $f(a) \subseteq g(a)$.

Theorem 2. *The sequence of bounded compositional semantics forms a monotonically increasing chain whose limit is identical to the compositional semantics, i.e.,* $\{\!|P|\!\}^0_{\mathrm{c}} \subseteq \{\!|P|\!\}^1_{\mathrm{c}} \subseteq \ldots \subseteq \bigcup_{k\in\mathbb{N}} \{\!|P|\!\}^k_{\mathrm{c}} = \{\!|P|\!\}_{\mathrm{c}}$.

Remark For simplicity, we present a compositional semantics in which information (i.e., the summary bags) only flows up from each sub-task to the task that posted it. However, an even more compact semantics—in the sense that more behaviors are expressed with the same bag-size bound—is possible when each task not only returns summaries, but also *receives* summaries as a parameter. For example, to interleave $2k$ summaries of two sub-tasks, one can pass the k summaries of the first sub-task to the second, and merge them with the second's summaries, as they are created, keeping only k at a time. Otherwise, one must interleave the two tasks' summaries outside, which means keeping $2k$ summaries for the enclosing task to interleave. Since the extension is purely technical, and not so insightful, we omit its description here. However, the results stated in the remainder of this paper are presented in terms of the bag-size bound w.r.t. this extension.

Note on Complexity. For the case where variables have finite-data domains, determining whether a global valuation is reachable is NP-complete[6] (in k) for

[6] Here the literature has assumed a fixed number of finite-state program variables in order to ignore the effect of complexity resulting from *data*. Indeed the reachability problem is EXPTIME-complete in the number of variables, due to the logarithmic encoding of states in the corresponding pushdown system.

the bounded-task k-context bounded and unbounded-task k-delay bounded sequentializations [20, 8], and PSPACE-complete (in k) for the unbounded-task k-context bounded sequentialization [17]. Since these cases are reducible to our semantics (see Section 5), it follows that global-state reachability is PSPACE-hard (in k) in the most general instance of our framework. Since the number of bag configurations is (singly) exponential in k, membership in EXPTIME is not hard to obtain. The practical difference between the NP/PSPACE-complete complexities of other sequentialization-based analyses is unclear; as sub-exponential time algorithms are not known for NP problems, exponential time is spent in the worst case, either way. Note however, that these complexity considerations are of limited significance, since we target an arbitrary class of sequential programs—not only programs with finite-data.

5 Global-Round Explorations

To understand the relationship between our bounded compositional exploration and the existing sequentializable analyses, we proceed in two steps. First we describe a restriction of our bounded semantics in which every task agrees on a global notion of execution "rounds," i.e., where each task executes (at most) one uninterrupted segment per global round. Second we show that this restriction captures and unifies the existing sequentializable analyses based on bounded context-switch and bounded delay.

A k *global-round execution* of a program P is an asynchronous execution of P where (i) each task executes in (up to) k uninterrupted segments called "rounds"—with the restriction that sub-tasks can only execute in, or after, the round in which it is created—and (ii) the i^{th} round of every task is executed before the $(i+1)^{\text{st}}$ round of *any* task, in the depth-first order over the task-creation tree; see Fig. 7a. Thus we can view each task as executing across a grid of k rounds, being interrupted $k-1$ times by the other tasks. With this view, each task can be characterized by an *interface* consisting of $2k$ global state valuations: the k global-state valuations seen by the task at the beginning of each round, and the k global-state valuations seen at the end (as in Fig. 7a). A task's interfaces are computed by guessing the initial global-state valuation of each round, following some number of sequential transitions, then nondeterministically advancing to the next round, keeping the computed local state, but updating the global-state to the next-guessed valuation. Given the interfaces for each task, sequentialization of the k global-round schedules is possible, by a reduction that executes each task once, according to the linear task-order, over a k-length vector of global-state valuations. The k *global-round semantics* of P, written $\{\!|P|\!\}^k_{\text{gr}}$ is the set of global valuations reachable in a k global-round execution of P.

Though we have defined the global-round semantics w.r.t. the asynchronous semantics, in fact we can restrict the k-bounded compositional semantics to compute only k global-round interfaces. During construction of the j^{th} round-summary of the current task t, the export-list contains $j-1$ summaries

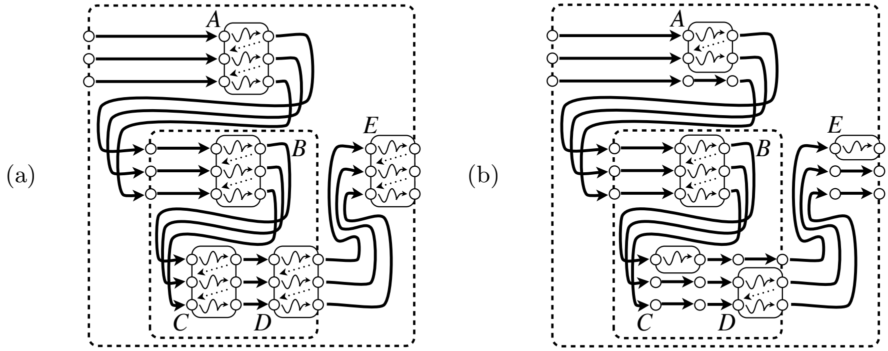


Fig. 7. (a) 3 global-round exploration, and (b) 4-delay exploration, for a program in which task A creates B and E, and task B creates C and D. Each task's interface (drawn with dashed-lines) summarizes 3 uninterrupted computations (drawn with squiggly arrows) of the task and all of its sub-tasks. Bold arrows connect global-state valuations, and dotted arrows connect local-state valuations. Note that the 3-round exploration allows each task 2 interruptions, while the 4-delay exploration allows all tasks combined only 4 interruptions.

(of rounds $1 \ldots j-1$) for t and its current sub-tasks, i.e., all descendants of t's thus-far posted tasks. At the same time, the bag maintains a $(k-j+1)$-length list of summaries accumulating the rounds $j \ldots k$ of t's current sub-tasks. Just before t begins round $j+1$, the accumulated summary of round j for t's current sub-tasks is consumed, so that the exported summary of round j captures the effect of one round of t, followed by one round of t's current sub-tasks. When t posts another task t_i (in round j), the $k-j+1$ summaries exported by t_i—note t_i and its descendants can only execute after round j of t—are combined with the $k-j+1$ accumulated summaries of t's current sub-tasks.[7] Thus when t completes round k, each round i summarizes round i of t followed by round i of each sub-task of t, in depth-first order, over the ordered task-creation tree of t.

Theorem 3. *The k global-round semantics is subsumed by the k-bounded compositional semantics, i.e., for all programs P we have* $\{\!|P|\!\}^k_{\text{gr}} \subsetneq \{\!|P|\!\}^k_{\text{c}}$.

In fact, our compositional semantics captures many more behaviors than k global-round analyses. For example, many behaviors cannot be expressed in k (global) rounds, though can be expressed by decoupling the rounds of different tasks. Intuitively, the additional power of our compositional semantics is in the ability to *reuse* the given bounded resources *locally*, ensuring only that the number of *visible* resources at any point is bounded. The full version of this paper [4] illustrates an example demonstrating this added analysis power.

[7] Using the extension described at the end of Section 4, this combination does not require additional bag space.

Theorem 4. *There exists a program P, $k_0 \in \mathbb{N}$, and a sequence $g_1, g_2, \ldots \in \{\!|P|\!\}_{\mathrm{c}}^{k_0}$ of global-state valuations such that for all $k \in \mathbb{N}$, $g_k \notin \{\!|P|\!\}_{\mathrm{gr}}^{k}$.*

5.1 Context-Bounding

In its initial formulation, the so-called "context-bounded" (CB) analyses explored the asynchronous executions of a program with a fixed number of statically allocated tasks, with at most two *context-switches* between tasks [23]. Shortly thereafter, Qadeer and Rehof [22] extended CB to explore an arbitrary bound k of context-switches.

Later, Lal and Reps [20] proposed a linear "round-robin" task-exploration order, and instead of bounding the number of context-switches directly, bounded the number of rounds in an explored round-robin schedule between n tasks. (It is easy to see that every k-context bounded execution with an unrestricted scheduler is also a k-round execution with a round-robin scheduler.) With this scheduling restriction, it becomes possible to summarize each task i's execution by an interface of k global-state valuation pairs, describing a computation that is interrupted by tasks $(i+1), (i+2), \ldots, n, 1, 2, \ldots, (i-1)$, $k-1$ times. In fact, this schedule is just a special case of the k global-round exploration, restricted to programs with a fixed number of statically-created tasks. La Torre et al. [17]'s subsequent extension of this k-round round-robin exploration to programs with a *parameterized* amount of statically-created tasks is also a special case of k global-round exploration, restricted to programs with an *arbitrary* number of statically-created tasks.

To formalize this connection, let a *static-task program* be an asynchronous program P which does not contain **post** statements, and an initial configuration $\langle g, \langle \ell, s \rangle, \emptyset \rangle$ is *(resp., parameterized) static-task initial* when s is of the form

$$\textbf{post}\ p_1\ \texttt{()};\ \ldots;\ \textbf{post}\ p_n\ \texttt{()} \qquad \text{(resp., } \textbf{while} \star \textbf{do}\ \textbf{post}\ p\ \texttt{()}\text{).}$$

A *k-round (resp., parameterized) CB execution* of a static-task program P is an asynchronous execution of P from a (resp., parameterized) static-task initial configuration c_0, where the initially posted tasks are dispatched in a round robin fashion over k rounds. The *k-round (resp., parameterized) CB semantics* of P, written $\{\!|P|\!\}_{\mathrm{cb}}^{k}$ (resp., $\{\!|P|\!\}_{\mathrm{cb}^*}^{k}$) is defined, as before, as the set of global valuations reachable from c_0 in a k-round (resp., parameterized) CB execution of P.

Theorem 5. *The k-round (parameterized) CB semantics is equal to the k global-round semantics, i.e., for all static-task programs P we have $\{\!|P|\!\}_{\mathrm{cb}^{(*)}}^{k} = \{\!|P|\!\}_{\mathrm{gr}}^{k}$.*

5.2 Delay-Bounding

Emmi et al. [8]'s recently introduced delay-bounded (DB) exploration[8] expands on the capabilities of Lal and Reps [20]'s k-round CB exploration with its ability

[8] Since we are interested here in analyses amenable to sequentialization, we consider only the depth-first delay-bounded task-scheduler [8].

to analyze programs with dynamic task-creation (i.e., with arbitrary use of the **post** statement). Like CB, the DB exploration considers round-based executions with a linear task-exploration order, though with dynamic task-creation the order must be defined over the task-creation tree; DB traverses the tree depth-first.

In fact, Emmi et al. [8]'s delay-bounded semantics is a variation of the k global-round semantics which, for the same bound k, expresses many fewer asynchronous executions. In essence, instead of allowing each task $k-1$ interruptions, the budget of interruptions is bounded globally, over the entire asynchronous execution; see Fig. 7b. It follows immediately that each task executes across at most k rounds, in the same sense as in the k global-round semantics. Since the mechanism behind delay-bounding is not crucial to our presentation here, we refer the reader to Emmi et al. [8]'s original work for the precise definition of *k-delay executions.* The *k-delay semantics* of P, written $\{\!|P|\!\}^k_{\text{db}}$ is the set of global valuations reachable in a k-delay execution of P.

Theorem 6. *The k-delay semantics is subsumed by the k global-round semantics, i.e., for all programs P we have $\{\!|P|\!\}^k_{\text{db}} \subseteq \{\!|P|\!\}^k_{\text{gr}}$.*

However, like the separation between k-global round semantics and k bounded semantics, there are families of behaviors that can be expressed with a fixed number of global rounds, though cannot be expressed with a fixed number of delays: for instance, behaviors which requires an unbounded number of tasks be interrupted (once) cannot be expressed with any finite number of delays.

Theorem 7. *There exists a program P, $k_0 \in \mathbb{N}$, and a sequence $g_1, g_2, \ldots \in \{\!|P|\!\}^{k_0}_{\text{gr}}$ of global-state valuations such that for all $k \in \mathbb{N}$, $g_k \notin \{\!|P|\!\}^k_{\text{db}}$.*

5.3 Context-Bounding vs. Delay-Bounding

It follows from Theorems 5 and 6 that context-bounding simulates delay-bounding on static-task programs. In fact, we can also show that delay-bounding simulates context-bounding, for programs with a fixed number of tasks, by combining Theorem 5 with the following theorem.

Theorem 8. *For a fixed number n of tasks, the k global-round semantics is subsumed by the nk-delay semantics, i.e., for all static-task programs P with n-tasks we have $\{\!|P|\!\}^k_{\text{gr}} \subseteq \{\!|P|\!\}^{nk}_{\text{db}}$.*

However, by Theorems 5 and 7, delay-bounding cannot simulate k-round parameterized context-bounded executions, since no fixed number of delays can express the unbounded number of potential context-switches.

6 Related Work

The technique of reducing a concurrent program behaviors to sequential program behaviors has garnered much attention in recent years. Based on the notion of *context-bounding* [23, 22, 21], Lal and Reps [20] showed how to encode the bounded-round round-robin schedules of a concurrent program with statically-allocated tasks as a sequential program. La Torre et al. [15] gave a more efficient encoding—in the sense that unreachable global-state valuations are never explored—and later extended the approach to programs with an unbounded number of statically-allocated tasks [17]. Emmi et al. [8] have recently extended the basic insight of round-based scheduling to sequentialize programs with an unbounded number of dynamically-created tasks. Empirical evidence suggests such techniques are indeed useful for bug-finding [21, 19, 11, 17]. For a more thorough comparison of these sequentializations, see Section 5.

Recently Kidd et al. [14] have shown how to sequentialize priority preemptive scheduled programs, and Garg and Madhusudan [10] have proposed an overapproximating sequentialization, albeit by exposing task-local state to other tasks. Both reductions assume a finite number of statically-declared tasks.

More generally, sequentialization could be seen as any linear traversal of the task-creation tree. The sequentializations we consider here are restricted to depth-first traversal, since they target sequential recursive programs, whose unbounded structure is, in general, contained to the procedure stack; the stack-based data-structure used for the depth-first traversal can be combined with the program's procedure stack. However, if one is willing to target other program models, one can consider other task-tree traversals, e.g., breadth-first using queues, or a completely non-deterministic traversal respecting the task-creation order, keeping, for instance, a multiset of horizon tasks. Atig et al. [1, 2]'s bounded explorations of programs with dynamic task-creation, by reduction to Petri-net reachability, are sequentializable in this sense.

Our characterization of sequentializability could also be relaxed to allow the exchange of local-state valuations (or alternatively, unbounded sequences of global-state valuations) between tasks. For instance, explorations based on *bounded languages* [13, 9] take this approach, essentially restricting concurrent exploration to inter-task interactions conforming to a regular *pattern*; then each task is executed in isolation by taking its product with the pattern-automaton. We simply note that the existing sequentializations avoid the (possibly expensive) computation of such a product.

7 Conclusion

We have proposed a framework for parameterized and compositional concurrent program analysis based on reduction to sequential program analysis. Our framework applies to a general class of shared-memory concurrent programs, with arbitrary preemption and dynamic task-creation, and strictly captures the known (round-based) sequentializations. It is our hope that this understanding will lead to further advances in sequential reductions, e.g., by enlightening efficiently-encodable instances of the general framework.

Though we have unified the existing sequentializations while maintaining their desirable qualities (i.e., sequential program model, compositionality, parameterization) by relaxing the global round-robin schedule, we are aware of one remaining restriction imposed by our framework. Besides the round-robin restriction imposed by the existing sequentializations, there is an implicit restriction imposed by translating task-creation directly to procedure calls: the tasks created by a single task are visited in the order they are created. Though further generalization is possible (e.g., by adding unbounded counters to the target program), such reductions will likely lead to much more complex analyses.

References

[1] Atig, M.F., Bouajjani, A., Touili, T.: Analyzing asynchronous programs with preemption. In: FSTTCS 2008: Proc. IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science. LIPIcs, vol. 2, pp. 37–48. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2008)

[2] Atig, M.F., Bouajjani, A., Qadeer, S.: Context-bounded analysis for concurrent programs with dynamic creation of threads. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 107–123. Springer, Heidelberg (2009)

[3] Ball, T., Rajamani, S.K.: The slam project: debugging system software via static analysis. In: POPL 2002: Proc. 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 1–3. ACM, New York (2002)

[4] Bouajjani, A., Emmi, M., Parlato, G.: On sequentializing concurrent programs (2011), http://hal.archives-ouvertes.fr/hal-00597415/en/

[5] Chaudhuri, S.: Subcubic algorithms for recursive state machines. In: POPL 2008: Proc. 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 159–169. ACM, New York (2008)

[6] Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL 1977: Proc. 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 238–252. ACM, New York (1977)

[7] DeLine, R., Leino, K.R.M.: BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research (2005)

[8] Emmi, M., Qadeer, S., Rakamarić, Z.: Delay-bounded scheduling. In: POPL 2011: Proc. 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 411–422. ACM, New York (2011)

[9] Ganty, P., Majumdar, R., Monmege, B.: Bounded underapproximations. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 600–614. Springer, Heidelberg (2010)

[10] Garg, P., Madhusudan, P.: Compositionality entails sequentializability. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 26–40. Springer, Heidelberg (2011)

[11] Ghafari, N., Hu, A.J., Rakamarić, Z.: Context-bounded translations for concurrent software: An empirical evaluation. In: van de Pol, J., Weber, M. (eds.) Model Checking Software. LNCS, vol. 6349, pp. 227–244. Springer, Heidelberg (2010)

[12] Jannet, B., Miné, A.: The Interproc analyzer, http://pop-art.inrialpes.fr/interproc/interprocweb.cgi

[13] Kahlon, V.: Tractable dataflow analysis for concurrent programs via bounded languages, Patent WO/2009/094439 (July 2009)

[14] Kidd, N., Jagannathan, S., Vitek, J.: One stack to run them all: Reducing concurrent analysis to sequential analysis under priority scheduling. In: van de Pol, J., Weber, M. (eds.) Model Checking Software. LNCS, vol. 6349, pp. 245–261. Springer, Heidelberg (2010)

[15] La Torre, S., Madhusudan, P., Parlato, G.: Reducing context-bounded concurrent reachability to sequential reachability. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 477–492. Springer, Heidelberg (2009)

[16] La Torre, S., Madhusudan, P., Parlato, G.: Analyzing recursive programs using a fixed-point calculus. In: PLDI 2009: Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 211–222. ACM, New York (2009)

[17] La Torre, S., Madhusudan, P., Parlato, G.: Model-checking parameterized concurrent programs using linear interfaces. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 629–644. Springer, Heidelberg (2010)

[18] Lahiri, S.K., Qadeer, S.: Back to the future: revisiting precise program verification using smt solvers. In: POPL 2008: Proc. 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 171–182. ACM, New York (2008)

[19] Lahiri, S.K., Qadeer, S., Rakamarić, Z.: Static and precise detection of concurrency errors in systems code using SMT solvers. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 509–524. Springer, Heidelberg (2009)

[20] Lal, A., Reps, T.W.: Reducing concurrent analysis under a context bound to sequential analysis. Formal Methods in System Design 35(1), 73–97 (2009)

[21] Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: PLDI 2007: Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 446–455. ACM, New York (2007)

[22] Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005)

[23] Qadeer, S., Wu, D.: KISS: Keep it simple and sequential. In: PLDI 2004: Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 14–24. ACM, New York (2004)

[24] Reps, T.W., Horwitz, S., Sagiv, S.: Precise interprocedural dataflow analysis via graph reachability. In: POPL 1995: Proc. 22th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 49–61. ACM, New York (1995)

[25] Reps, T.W., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. Sci. Comput. Program. 58(1-2), 206–263 (2005)

Verifying Fence Elimination Optimisations

Viktor Vafeiadis[1] and Francesco Zappa Nardelli[2]

[1] MPI-SWS
[2] INRIA

Abstract. We consider simple compiler optimisations for removing redundant memory fences in programs running on top of the x86-TSO relaxed memory model. While the optimisations are performed using standard thread-local control flow analyses, their correctness is subtle and relies on a non-standard global simulation argument. The implementation and the proof of correctness are programmed in Coq as part of CompCertTSO, a fully-fledged certified compiler from a concurrent extension of a C-like language to x86 assembler. In this article, we describe the soundness proof of the optimisations and evaluate their effectiveness.

1 Introduction

Contrary to a naïve programmer's expectations, modern multicore architectures do not implement a sequentially consistent (SC) shared memory, but rather exhibit a relaxed consistency model. For instance, x86 provides a TSO-like memory model [20,25] while Power exhibits much more relaxed behaviour [21].

Programming directly against such relaxed hardware can be difficult, especially if performance and portability are a concern. For this reason, programming languages need their own higher-level memory model, and it is the role of the compiler to guarantee that the language memory model is correctly implemented on the target hardware architecture. Naturally, SC comes as an attractive choice because it is intuitive to programmers, and SC behaviour for volatiles and SC atomics is an important part of the Java and C++0x memory models, respectively. However, implementing SC over relaxed hardware memory models requires the insertion of potentially expensive memory fence instructions, and if done naïvely results in a significant performance degradation. For instance, if SC is recovered on x86 by systematically inserting an `MFENCE` instruction either after every memory store instruction, or before every memory load (as in a prototype implementation of C++0x atomics [28]), then on some benchmarks (e.g. Fraser's library of lock-free algorithms [12]) we could observe a 40% slowdown with respect to a hand-optimised implementation.

Compiler optimisations to optimise barrier placement are thus essential, and indeed there are many opportunities to perform fence removal optimisations. For example, on x86, if there are no writes between two memory fence instructions, the second fence is unnecessary. Dually, if there are no reads between the two fence instructions, then the first fence instruction is redundant. Finally, by a form of partial redundancy elimination [19], we can insert memory

E. Yahav (Ed.): SAS 2011, LNCS 6887, pp. 146–162, 2011.

barriers at selected program points in order to make other fence instructions redundant, with an overall effect of hoisting barriers out of loops and reducing the number of fences along some execution paths without ever increasing it on any path.

However, concurrency, especially relaxed-memory concurrency, is a notoriously subtle and error-prone domain [11,22], and so *verifying* such optimisations is of great interest. Work in this area goes back at least to that of Shasha and Snir [26] (and more recently [2,6,23]), but most of this is in terms of transformations of hypothetical program executions rather than the transformations of code that are implemented (without proof) in actual compilers.

In this paper, we bridge this gap by implementing the aforementioned redundant barrier removal optimisations in CompCertTSO [24], a fully fledged verified compiler from ClightTSO (a C-like concurrent language with **T**otal **S**tore **O**rder semantics) to concurrent x86 assembly code with x86-TSO semantics. We prove the correctness of our optimisations in Coq and integrate this result into the overall semantic preservation proof of CompCertTSO, giving an end-to-end correctness result.[1] The correctness result verifies that these fence removal optimisations do not introduce any new TSO behaviour. They are therefore sound in several different usages: for ClightTSO code with manually inserted fences; in a TSO implementation of a DRF-based memory model, such as C++0x [3] or the Java memory model [18], where fences are used to implement the language's low-level atomic primitives; or if one implements SC by starting with a placement of fences admitting only SC-behaviours (e.g., by placing a fence after every memory write) and then optimises away as many as possible.

The correctness of one of our optimisations turned out to be more much interesting than we had anticipated and could not be verified using a standard forward simulation [17] because it introduces unobservable non-determinism (see §3 for an explanation). To verify this optimisation, we introduce *weaktau simulations* (see §4), which in our experience were much easier to use than backward simulations [17]. In contrast, the other two optimisations were straightforward to verify, each taking us less than a day's worth of work to prove correct in Coq.

Hence, we believe that developing mechanised formal proofs about concurrent optimising compilers offers a good benefit to effort ratio, once the right foundations are laid out.

Outline. We begin, in §2, by recalling the relaxed-memory behaviour of our target architecture, as captured by the x86-TSO model, and the structure and correctness statement of the CompCertTSO verified compiler. Then, in §3, we describe our optimisations and their implementation in CompCertTSO, and evaluate their performace. In §4, we describe our overall simulation proof strategy for verifying compiler correctness, and in §5 the use of this strategy to verify the optimisations in question. Finally, in §6, we reflect on our experience using Coq and in §7 we discuss related work.

[1] The proofs are available at `http://www.cl.cam.ac.uk/~pes20/CompCertTSO/`.

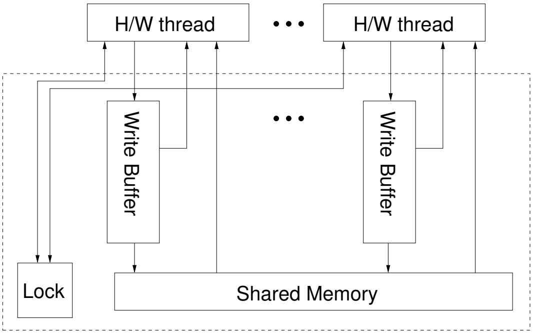


Fig. 1. x86-TSO block diagram

2 The x86-TSO Memory Model and CompCertTSO

The x86-TSO model [20,25] is a rigorous memory model that captures the memory behaviour of the x86 architecture visible to the programmer, for normal code. Figure 1 depicts the x86-TSO block diagram: each hardware thread has a FIFO buffer of pending memory writes (which can propagate to memory at any later time, thereby avoiding the need to block while a write completes); memory reads return the latest memory write pending in the thread write buffer, or, if there are no pending writes, the value written in the shared memory. Memory fences (`MFENCE`) flush the local write buffer: formally, they block until the buffer is empty. 'Locked' x86 instructions (e.g. `LOCK INC`) involve multiple memory accesses performed atomically. Such atomic read-modify-write instructions first block until the local write buffer is empty, and then atomically perform a read, a write of the updated value, and a flush of the local write buffer.

The classic example showing a non-SC behaviour in TSO is the store buffer program below (SB). Given two distinct memory locations `x` and `y` initially holding `0`, the memory writes of two hardware threads respectively writing `1` to `x` and `y` can be buffered, and the subsequent reads from `y` and `x` (into register `EAX` on thread 0 and `EBX` on thread 1) are fulfilled by the memory (which still holds `0` everywhere), and it is possible for both to read `0` in the same execution. It is easy to check that this result cannot arise from any SC interleaving, but it is observable on modern Intel or AMD x86 multiprocessors.

If `MFENCE` instructions are inserted after the memory writes, as in SB+mfences, then at least one buffer must be flushed before any read is performed, and at least one thread will observe the write performed by the other thread, as in all SC executions.

SB

Thread 0	Thread 1
MOV [x]←1 MOV EAX←[y]	MOV [y]←1 MOV EBX←[x]
Allowed final state: 0:EAX=0 ∧ 1:EBX=0	

SB+mfences

Thread 0	Thread 1
MOV [x]←1 MFENCE MOV EAX←[y]	MOV [y]←1 MFENCE MOV EBX←[x]
Forbidden final state: 0:EAX=0 ∧ 1:EBX=0	

CompCertTSO [**24**,**9**] is a certified compiler that lifts the x86-TSO model of the x86 assembly language to a C-like language. It builds on CompCert [16]. Its source language, ClightTSO, is a concurrent extension of CompCert Clight language [4], adding thread creation and some atomic read-modify-write and barrier primitives that are directly implementable by x86 locked instructions and `MFENCE`. In addition, ClightTSO load and store operations have a TSO semantics. The main syntactic difference between Clight and C is that expressions cannot contain function calls and memory writes (these can occur only as statements).

The behaviour of the source and target languages (ClightTSO and x86-TSO) is defined using labelled transition systems (LTS) with visible actions for call and return of external functions (e.g. OS I/O primitives), program exit, semantic failure, and out-of-memory error, together with internal τ actions.[2]

$$\mathit{event},\ \mathit{ev} ::= \texttt{call}\,\mathit{id}\ \mathit{vs} \mid \texttt{return}\,\mathit{typ}\ v \mid \texttt{exit}\,n \mid \texttt{fail} \mid \texttt{oom} \mid \tau$$

We take the observable behaviours of a program to be the set of finite and infinite traces of events it generates, filtering out any finite sequences of internal τ actions. Broadly speaking, the correctness property of CompCertTSO states that if the compiled program has some observable behaviours then those behaviours are admitted by the source semantics; however compiled behaviour that arises from an erroneous source program need not to be admitted in the source semantics, and the compiled program should only diverge, indicated by an infinite trace of τ labels, if the source program can. This is formalised in §4.

Adapted from CompCert 1.5 [8], the compilation from ClightTSO to x86-TSO goes through 13 successive phases and 7 intermediate languages (Csharpminor, Cminor, RTL, LTL, LTLin, Linear, Mach), which progressively transform C features into assembly code and perform various standard optimisations such as constant propagation, CSE (limited to arithmetic expressions), branch tunneling, and register allocation. In this paper, we need consider only the RTL intermediate language, whose programs consist of a set of function definitions each containing a control flow graph (CFG) of three-address-code instructions:

$$\begin{aligned}\mathit{rtl\_instr} ::= {} & \texttt{nop} \mid \texttt{op}(\mathit{op}, \vec{r}, r) \mid \texttt{load}(\kappa, \mathit{addr}, \vec{r}, r) \mid \texttt{store}(\kappa, \mathit{addr}, \vec{r}, \mathit{src}) \\ & \mid \texttt{call}(\mathit{sig}, \mathit{ros}, \mathit{args}, \mathit{res}) \mid \texttt{cond}(\mathit{cond}, \mathit{args}) \mid \texttt{return}(\mathit{optarg}) \\ & \mid \texttt{threadcreate}(\mathit{optarg}) \mid \texttt{atomic}(\mathit{aop}, \vec{r}, r) \mid \texttt{fence}\end{aligned}$$

Nodes with `cond` instructions have two successors (*ifso*, *ifnot*); nodes with `return` instructions have no successors; all other nodes have exactly one successor.

3 The Optimisations

We detect and optimise away the following cases of redundant `MFENCE` instructions:

- a fence is redundant if it always follows a previous fence or locked instruction in program order, and no memory store instructions are in between (FE1);

[2] Internal actions include local computations, memory accesses and TSO-unbufferings.

– a fence is redundant if it always precedes a later fence or locked instruction in program order, and no memory read instructions are in between (FE2).

We also perform partial redundancy elimination (PRE) [19] to improve on the second optimisation: we selectively insert memory fences in the program to make fences that are redundant along *some* execution paths to be redundant along all paths, which allows FE2 to eliminate them. The combined effect of PRE and FE2 is quite powerful and can even hoist a fence instruction out of a loop, as we shall see later in this section.

The correctness of FE1 is intuitive: since no memory writes have been performed by the same thread since executing an atomic instruction, the thread's buffer must be empty and so the fence instruction is effectively a no-op and can be optimised away.

The correctness of FE2 is more subtle. To see informally why it is correct, first consider the simpler transformation that swaps a MFENCE instruction past an adjacent store instructions (that is, MFENCE;store ⇝ store;MFENCE). To a first approximation, we can think of FE2 as successively applying this transformation to the earlier fence (and also commuting it over local non-memory operations) until it reaches the later fence; then we have two successive fences and we can remove one. Intuitively, the end-to-end behaviours of the transformed program, store;MFENCE, are a subset of the end-to-end behaviours of the original program, MFENCE;store: the transformed program leaves the buffer empty, whereas in the original program there can be up to one outstanding write in the buffer. Notice that there is an intermediate state in the transformed program that is not present in the original program: if initially the buffer is non-empty, then after executing the store instruction in store;MFENCE we end up in a state where the buffer contains the store and some other elements. It is, however, impossible to reach the same state in the original MFENCE;store program because the store always goes into an empty buffer. What saves soundness is that this intermediate state is not observable. Since threads can access only their own buffers, the only way to distinguish an empty buffer from a non-empty buffer must involve the thread performing a read instruction from that intermediate state.

Indeed, if there are any intervening reads between the two fences, the transformation is unsound, as illustrated by the following variant of SB+mfences:

Thread 0	Thread 1
MOV [x]←1	MOV [y]←1
MFENCE (*)	MFENCE
MOV EAX←[y]	MOV EBX←[x]
MFENCE	

If the MFENCE labelled with (*) is removed, then it is easy to find an x86-TSO execution that terminates in a state where EAX and EBX are both 0, which was impossible in the unoptimised program.

This 'swapping' argument works for finite executions, but does not account for infinite executions, as it is possible that the later fence is never executed — if, for example, the program is stuck in an infinite loop between the two fences.

$$\begin{array}{ll@{\qquad}ll}
T_1(\mathtt{nop}, \mathcal{E}) & = \mathcal{E} & T_2(\mathtt{nop}, \mathcal{E}) & = \mathcal{E} \\
T_1(\mathtt{op}(op, \vec{r}, r), \mathcal{E}) & = \mathcal{E} & T_2(\mathtt{op}(op, \vec{r}, r), \mathcal{E}) & = \mathcal{E} \\
T_1(\mathtt{load}(\kappa, addr, \vec{r}, r), \mathcal{E}) & = \mathcal{E} & T_2(\mathtt{load}(\kappa, addr, \vec{r}, r), \mathcal{E}) & = \top \\
T_1(\mathtt{store}(\kappa, addr, \vec{r}, src), \mathcal{E}) & = \top & T_2(\mathtt{store}(\kappa, addr, \vec{r}, src), \mathcal{E}) & = \mathcal{E} \\
T_1(\mathtt{call}(sig, ros, args, res), \mathcal{E}) & = \top & T_2(\mathtt{call}(sig, ros, args, res), \mathcal{E}) & = \top \\
T_1(\mathtt{cond}(cond, args), \mathcal{E}) & = \mathcal{E} & T_2(\mathtt{cond}(cond, args), \mathcal{E}) & = \mathcal{E} \\
T_1(\mathtt{return}(optarg), \mathcal{E}) & = \top & T_2(\mathtt{return}(optarg), \mathcal{E}) & = \top \\
T_1(\mathtt{threadcreate}(optarg), \mathcal{E}) & = \top & T_2(\mathtt{threadcreate}(optarg), \mathcal{E}) & = \top \\
T_1(\mathtt{atomic}(aop, \vec{r}, r), \mathcal{E}) & = \bot & T_2(\mathtt{atomic}(aop, \vec{r}, r), \mathcal{E}) & = \bot \\
T_1(\mathtt{fence}, \mathcal{E}) & = \bot & T_2(\mathtt{fence}, \mathcal{E}) & = \bot
\end{array}$$

Fig. 2. Transfer functions for FE1 and FE2

The essential difficulty of the proof is that FE2 introduces non-observable nondeterminism. It is well-known that reasoning about such transformations cannot, in general, be done solely by a standard forward simulation (e.g., [17]), but it also requires a backward simulation [17] or, equivalently, prophecy variables [1]. We have tried using backward simulation to carry out the proof, but found the backward reasoning painfully difficult. Instead, we came up with a new kind of forward simulation, which we call a *weaktau simulation* in §4, that incorporates a simple version of a boolean prophecy variable that is much easier to use and suffices to verify FE2. The details are in §4 and §5.

We can observe that neither optimisation subsumes the other: in the program below on the left the (*) barrier is removed by FE2 but not by FE1, while in the program on the right the (†) barrier is removed by FE1 but not by FE2.

```
MOV [x]←1          MFENCE
MFENCE (*)         MOV EAX←[x]
MOV [x]←2          MFENCE (†)
MFENCE             MOV EBX←[y]
```

Implementation. The fence instructions eligible to be optimised away are easily computed by two intra-procedural dataflow analyses over the boolean domain, $\{\bot, \top\}$, performed on RTL programs. Among the intermediate languages of CompCertTSO, RTL is the most convenient to perform these optimisations, and it is the intermediate language where most of the existing optimisations are performed: namely, constant propagation, CSE, and register allocation.

The first is a *forward* dataflow problem that associates to each program point the value $\bot$ if along *all* execution paths there is an atomic instruction *before* the current program point with no intervening writes, and $\top$ otherwise. The problem can be formulated as the solution of the standard forward dataflow equation:

$$\mathcal{FE}_1(n) = \begin{cases} \top & \text{if predecessors}(n) = \emptyset \\ \bigsqcup_{p \in \text{predecessors}(n)} T_1(instr(p), \mathcal{FE}_1(p)) & \text{otherwise} \end{cases}$$

where p and n are program points (i.e., nodes of the control-flow-graph), the join operation is logical disjunction (returning $\top$ if at least one of the arguments is $\top$), and the transfer function T_1 is defined in Fig. 2.

The second is a *backward* dataflow problem that associates to each program point the value $\bot$ if along *all* execution paths there is an atomic instruction *after* the current program point with no intervening reads, and $\top$ otherwise. This problem is solved by the standard backward dataflow equation:

$$\mathcal{FE}_2(n) = \begin{cases} \top & \text{if successors}(n) = \emptyset \\ \bigsqcup_{s \in \text{successors}(n)} T_2(\mathit{instr}(s), \mathcal{FE}_2(s)) & \text{otherwise} \end{cases}$$

where the join operation is again logical disjunction and the transfer function T_2 is defined in Fig. 2.

To solve the dataflow equations we reuse the generic implemenation of Kildall's algorithm provided by the CompCert compiler. Armed with the results of the dataflow analysis, a pass over the RTL source replaces the fence nodes whose associated value in the corresponding analysis is $\bot$ with `nop` (no-operation) nodes, which are removed by a later pass of the compiler.

Partial Redundancy Elimination. In practice, it is common for `MFENCE` instructions to be redundant on some but not all paths through a program. To help with these cases, we perform a partial redundancy elimination phase (PRE) that inserts `fence` instructions so that partially redundant fences become fully redundant. For instance, the RTL program on the left of Fig. 3 (from Fraser's `lockfree-lib`) cannot be optimised by FE2: PRE inserts a memory fence in the *ifnot* branch, which in turn enables FE2 to rewrite the program so that all execution paths go through at most one `fence` instruction.

The implementation of PRE runs two static analyses to identify the program points where `fence` nodes should be introduced. First, the RTL generation phase introduces a nop as the first node on each branch after a conditional; these nop nodes will be used as placeholders to insert (or not) the redundant barriers. We then run two static analyses:

– the first, called A, is a backward analysis returning $\top$ if along *some* path after the current program point there is an atomic instruction with no intervening reads;
– the second, called B, is a forward analysis returning $\bot$ if along *all* paths to the current program point there is a `fence` with no later reads or atomic instructions.

The transformation inserts fences *after conditional nodes* on branches whenever:

– analysis B returns $\bot$ (i.e., there exists a previous fence that will be eliminated if we were to insert a fence at both branches of the conditional nodes); and
– analysis A returns $\bot$ (i.e., the previous `fence` will not be removed by FE2); and
– analysis A returns $\top$ *on the other branch* (the other branch of the conditional already makes the previous `fence` partially redundant).

If all three conditions hold for a `nop` node following a branch instruction, then that node is replaced by a `fence` node. A word to justify the *some path* (instead

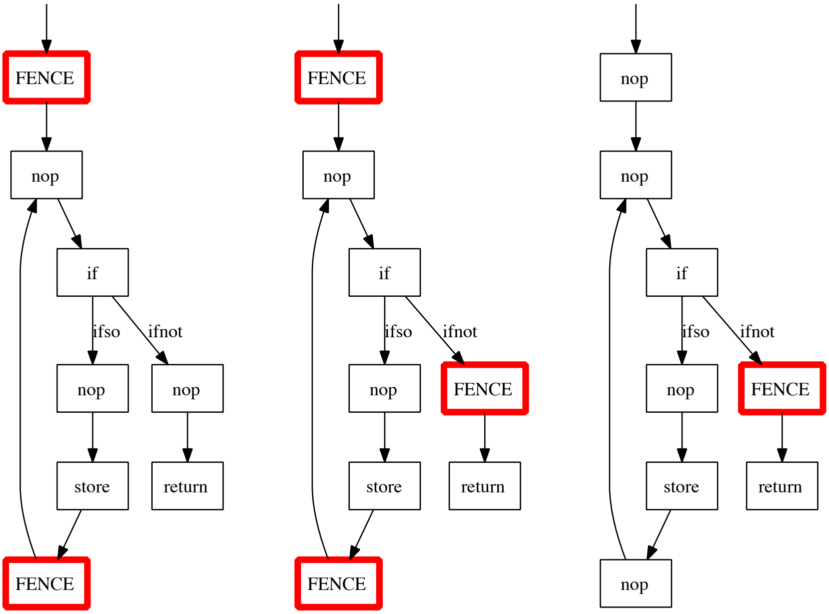


Fig. 3. Unoptimised RTL, RTL after PRE, and RTL after PRE and FE2

of *for all paths*) condition in analysis A: as long as there is a fence on some path, then at all branch points PRE would insert a fence on all other paths, essentially converting the program to one having fences on all paths.

The transfer functions T_A and T_B are detailed in Fig. 4. Note that T_B defines the same transfer function as T_2, but here it is used in a forward, rather than backward, dataflow problem.

Evaluation. We instructed the RTL generation phase of CompCertTSO to systematically introduce a `MFENCE` instruction before each memory read (strategy *br*), or after each memory write (strategy *aw*). The table in Figure 5 considers several well-known concurrent algorithms, including Dekker and Bakery mutual exclusion algorithms, Treiber's stack [29], the TL2 lock-based STM [10], the already mentioned Fraser's lockfree implementation of skiplists, and several benchmarks from the STAMP benchmark [7], and reports the total numbers of fences in the generated assembler files, following the *br* and *aw* strategies, possibly enabling the FE1, PRE and FE2 optimisations.

A basic observation is that FE2 removes on average about 30% of the `MFENCE` instructions, while PRE does not further reduce the static number of fences, but rather reduces the dynamic number of fences executed, e.g. by hoisting fences out of loops as in Figure 3. When it comes to execution times, then the gain is much more limited than the number of fences removed. For example, we observe a 3% speedup when PRE and FE2 are used on the skiplist code (running `skiplist 2`

$$\begin{array}{ll@{\qquad}ll}
T_A(\texttt{nop}, \mathcal{E}) & = \mathcal{E} & T_B(\texttt{nop}, \mathcal{E}) & = \mathcal{E} \\
T_A(\texttt{op}(\mathit{op}, \vec{r}, r), \mathcal{E}) & = \mathcal{E} & T_B(\texttt{op}(\mathit{op}, \vec{r}, r), \mathcal{E}) & = \mathcal{E} \\
T_A(\texttt{load}(\kappa, \mathit{addr}, \vec{r}, r), \mathcal{E}) & = \bot & T_B(\texttt{load}(\kappa, \mathit{addr}, \vec{r}, r), \mathcal{E}) & = \top \\
T_A(\texttt{store}(\kappa, \mathit{addr}, \vec{r}, \mathit{src}), \mathcal{E}) & = \mathcal{E} & T_B(\texttt{store}(\kappa, \mathit{addr}, \vec{r}, \mathit{src}), \mathcal{E}) & = \mathcal{E} \\
T_A(\texttt{call}(\mathit{sig}, \mathit{ros}, \mathit{args}, \mathit{res}), \mathcal{E}) & = \bot & T_B(\texttt{call}(\mathit{sig}, \mathit{ros}, \mathit{args}, \mathit{res}), \mathcal{E}) & = \top \\
T_A(\texttt{cond}(\mathit{cond}, \mathit{args}), \mathcal{E}) & = \mathcal{E} & T_B(\texttt{cond}(\mathit{cond}, \mathit{args}), \mathcal{E}) & = \mathcal{E} \\
T_A(\texttt{return}(\mathit{optarg}), \mathcal{E}) & = \bot & T_B(\texttt{return}(\mathit{optarg}), \mathcal{E}) & = \top \\
T_A(\texttt{threadcreate}(\mathit{optarg}), \mathcal{E}) & = \bot & T_B(\texttt{threadcreate}(\mathit{optarg}), \mathcal{E}) & = \top \\
T_A(\texttt{atomic}(\mathit{aop}, \vec{r}, r), \mathcal{E}) & = \top & T_B(\texttt{atomic}(\mathit{aop}, \vec{r}, r), \mathcal{E}) & = \bot \\
T_A(\texttt{fence}, \mathcal{E}) & = \top & T_B(\texttt{fence}, \mathcal{E}) & = \bot
\end{array}$$

Fig. 4. Transfer functions for analyses A and B of PRE

	br	*br*+FE1	*aw*	*aw*+FE2	*aw*+PRE+FE2
Dekker	3	2	5	4	4
Bakery	10	2	4	3	3
Treiber's stack	5	2	3	1	1
Fraser's skiplist	32	18	19	12	11
TL2	166	95	101	68	68
Genome	133	79	62	41	41
Labyrinth	231	98	63	42	42
SSCA	1264	490	420	367	367

Fig. 5. Experimental results

`50 100` on a 2-core x86 machine): the hand-optimised (barrier free) version by Fraser is about 45% faster than the code generated by the *aw* strategy.

For Lamport's bakery algorithm we generate optimal code for `lock`, as barriers are used to restore SC on accesses to the `choosing` array. However the particular optimisations we consider are clearly not the last word on the subject. Looking at the fences we do not remove in more detail, the Treiber stack is instructive, as the only barrier left corresponds to an update to a newly allocated object, and our analyses cannot guess that this newly allocated object is still local; a precise escape analysis would be required. In general, about the half of the remaining `MFENCE` instructions precede a function call or return; we believe that performing an interprocedural analysis would remove most of these barriers. Our focus here is on *verified* optimisations rather than performance alone, and the machine-checked correctness proof of such sophisticated optimisations is a substantial challenge for future work.

4 Formalisation of Traces and Simulations

In this section, we formalise the traces of a program, the correctness statement for our compiler, as well as basic, measured, and weaktau simulations. This section corresponds to the Coq file `Traces.v` in our distribution [9], and was not part of our original work on CompCertTSO [24], where we took measured simulations to be our compiler correctness statement.

Language Semantics. Abstractly, the operational semantics of a language such as ClightTSO, RTL, and x86-TSO consists of a type of programs, *prg* and a type of states, *states*, together with a set of initial states for each program, $\mathsf{init} \in \mathit{prg} \to \mathbb{P}(\mathit{states})$, and a transition relation, $\to \in \mathbb{P}(\mathit{states} \times \mathit{event} \times \mathit{states})$. The states contain the program text, the memory, the buffers for each thread, and each thread's local state (for RTL, this is the program counter, the stack pointer, and the values of the registers).

We employ the following notations: (i) $s \xrightarrow{ev} s'$ stands for $(s, ev, s') \in\to$; (ii) $s \not\to$ means $\neg(\exists ev\, s'.\ s \xrightarrow{ev} s')$; and (iii) $s \xrightarrow{\tau}{}^{*} \xrightarrow{ev} s'$ means $\exists s''.\ s \xrightarrow{\tau}{}^{*} s'' \wedge s'' \xrightarrow{ev} s'$. For a finite sequence ℓ of non-τ, non-`oom`, non-`fail` events, we define $s \overset{\ell}{\Rightarrow} s'$ to hold whenever s can do the sequence ℓ of events, possibly interleaved with a finite number of τ-events, and end in state s'. Further, we define the predicate $\mathsf{inftau}(s)$ to hold whenever s can do an infinite sequence of τ-transitions.

Traces. Traces are either infinite sequences of non-τ events or finite sequences of non-τ events ending with one of the following three placeholders: end (designating successful termination), inftau (designating an infinite execution that eventually stops performing any visible events), or oom (designating an execution that ends because it runs out of memory). The traces of a program, p, are given as follows:

$$\begin{aligned}
\mathsf{traces}(p) \overset{\text{def}}{=}\ & \{\ell \cdot \mathsf{end} \mid \exists s \in \mathsf{init}(p).\ \exists s'.\ s \overset{\ell}{\Rightarrow} s' \wedge s' \not\to\} \\
& \cup \{\ell \cdot tr \mid \exists s \in \mathsf{init}(p).\ \exists s'.\ s \xRightarrow{\ell\cdot\mathtt{fail}} s'\} \\
& \cup \{\ell \cdot \mathsf{inftau} \mid \exists s \in \mathsf{init}(p).\ \exists s'.\ s \overset{\ell}{\Rightarrow} s' \wedge \mathsf{inftau}(s')\} \\
& \cup \{\ell \cdot \mathsf{oom} \mid \exists s \in \mathsf{init}(p).\ \exists s'.\ s \overset{\ell}{\Rightarrow} s'\} \\
& \cup \{tr \mid \exists s \in \mathsf{init}(p).\ s \text{ can do the infinite trace } tr\}
\end{aligned}$$

We treat failed computations as having arbitrary behaviour after their failure point, whereas we allow the program to run out of memory at any point during its execution. This perhaps counter-intuitive semantics of oom is needed to get a correctness statement guaranteeing nothing about computations that run out of memory.

Simulations. We proceed to the definition of simulations, which are techniques for proving statements of the form $\forall p.\ \mathsf{traces}(\mathsf{compile}(p)) \subseteq \mathsf{traces}(p)$, which we consider as $\mathsf{compile}$'s correctness statement.

Definition 1 (Basic sim.). *The relation pair* $\sim \in \mathbb{P}(\mathit{src.states} \times \mathit{tgt.states})$ *and* $> \in \mathbb{P}(\mathit{tgt.states} \times \mathit{tgt.states})$ *is a* basic simulation *for the compilation function* $\mathsf{compile} : \mathit{src.prg} \to \mathit{tgt.prg}$*, if and only if the following properties are satisfied.*[3]

[3] Our Coq definition in `Traces.v` exploits some particular properties common to all our semantics (e.g., $s \not\to$ if and only if s contains no threads), and is therefore slightly different than the one presented in the paper.

$$\begin{array}{ll}
\mathit{sim\_init} : \forall p\, p'.\ \mathsf{compile}(p) = p' \implies \forall t \in \mathsf{init}(p').\ \exists s \in \mathsf{init}(p).\ s \sim t \\
\mathit{sim\_end} : \forall s\, t.\ s \sim t \wedge t \not\to \_ \implies s \not\to \_ \\
\mathit{sim\_step} : \forall s\, t\, t'\, ev.\ s \sim t \wedge t \xrightarrow{ev} t' \wedge ev \neq \mathsf{oom} \implies \\
\qquad\qquad (s \xrightarrow{\tau}{}^{*} \xrightarrow{\mathtt{fail}} \_) & \text{— } \textit{s reaches a failure} \\
\qquad\quad \vee\, (\exists s'.\ s \xrightarrow{\tau}{}^{*} \xrightarrow{ev} s' \wedge s' \sim t') & \text{— } \textit{s does matching step sequence} \\
\qquad\quad \vee\, (ev = \tau \wedge t > t' \wedge s \sim t')\,. & \text{— } \textit{s stutters (only allowed if } t > t'\textit{)}
\end{array}$$

In the definition above, $>$ is used simply to control when stuttering can occur. Later definitions will impose restrictions on $>$.

It is well-known (e.g., [17]) that exhibiting a basic simulation is sufficient to verify that the finite traces of $\mathsf{compile}(p)$ are included in the traces of p. This is captured by the following lemmata:

Lemma 1. *If* $(\sim, >)$ *is a basic simulation, then for all* s, t, t', *and* ev, *if* $s \sim t$ *and* $t \xrightarrow{\tau}{}^{*} \xrightarrow{ev} t'$ *and* $ev \neq \mathsf{oom}$, *then*

$$(s \xrightarrow{\tau}{}^{*} \xrightarrow{\mathtt{fail}} \_) \vee (\exists s'.\ s \xrightarrow{\tau}{}^{*} \xrightarrow{ev} s' \wedge s' \sim t') \vee (ev = \tau \wedge s \sim t').$$

Lemma 2. *If* $(\sim, >)$ *is a basic simulation, then for all* s, t, t', *and* tr, *if* $s \sim t$ *and* $t \overset{tr}{\Longrightarrow} t'$, *then* $(\exists tr_1\, tr_2.\ tr = tr_1 \cdot tr_2 \wedge s \xRightarrow{tr_1 \cdot \mathtt{fail}} \_) \vee (\exists s'.\ s \overset{tr}{\Longrightarrow} s' \wedge s' \sim t')$.

The second lemma says that given a trace tr of external events starting from a target state, t, then the related source state, s, can either fail after doing a prefix of the trace or can do the same trace and end in a related state. (This is proved by an induction on the length of the trace tr using Lemma 1.)

Using coinductive reasoning, we can extend this proof to cover infinite traces of external events. However, we cannot show something similar for a trace ending with infinite sequence of internal events, because the third disjunct of *sim_step* effectively allows us to stutter forever, thereby possibly removing the infinite sequence of internal events.

So, while basic simulations do not imply full trace inclusion, they do so if we can further show that for all $s \sim t$, $\mathsf{inftau}(t)$ implies $\mathsf{inftau}(s)$.

Lemma 3. *If there exists a basic simulation* $(\sim, >)$ *for the compilation function* $\mathsf{compile}$, *and if for all* $s \sim t$, $\mathsf{inftau}(t)$ *implies* $\mathsf{inftau}(s)$, *then for all programs* p, $\mathsf{traces}(\mathsf{compile}(p)) \subseteq \mathsf{traces}(p)$.

This theorem follows easily from Lemma 2 and the corresponding lemma for infinite traces of external events.

To ensure inclusion even for infinite traces of internal events, CompCertTSO uses *measured simulations*, which additionally require that $>$ is well-founded.

Definition 2 (Measured sim.). *A* measured simulation *is any basic simulation* $(\sim, >)$ *such that* $>$ *is well-founded.*

Existence of a measured simulation implies full trace inclusion, intuitively because we can no longer remove an infinite sequence of internal events.

Theorem 1. *If there exists a measured simulation for the compilation function* $\mathsf{compile}$*, then for all programs* p*,* $\mathsf{traces}(\mathsf{compile}(p)) \subseteq \mathsf{traces}(p)$*.*

In this work, we introduce a new kind of simulation: the *weaktau* simulation, which also implies trace inclusion.

Definition 3 (Weaktau sim.). *A* weaktau simulation *consists of a basic simulation* $(\sim, >)$ *with an additional relation between source and target states,* $\simeq \in \mathbb{P}(src.states \times tgt.states)$ *satisfying the following properties:*

$$\begin{aligned}
&\mathit{sim\_weaken} : \forall s, t.\ s \sim t \implies s \simeq t \\
&\mathit{sim\_wstep} : \forall s\,t\,t'.\ s \simeq t \wedge t \xrightarrow{\tau} t' \wedge t > t' \implies \\
&\qquad\qquad (s \xrightarrow{\tau}{}^{*} \xrightarrow{\texttt{fail}} \_) \qquad\qquad\qquad \textit{— s reaches a failure} \\
&\qquad\quad \vee (\exists s'.\ s \xrightarrow{\tau}{}^{*} \xrightarrow{\tau} s' \wedge s' \simeq t') \quad \textit{— s does a matching step sequence.}
\end{aligned}$$

One way of seeing weaktau simulations is as a forward simulation incorporating a boolean prophecy variable [1] that can be used to delay execution only of internal τ steps, but not of any visible steps. This will become more evident from the proof that weaktau simulations imply trace inclusion.

Theorem 2. *If there exists a weaktau-simulation* $(\sim, >, \simeq)$ *for the compilation function* $\mathsf{compile}$*, then for all programs* p*,* $\mathsf{traces}(\mathsf{compile}(p)) \subseteq \mathsf{traces}(p)$*.*

Proof (sketch). From Lemma 3, it suffices to prove that whenever $s \sim t$ and there is an infinite sequence of internal events starting from t, then there is also such a sequence starting from s. To construct such a trace, we do a case split: Are the transitions eventually always in the $>$ relation (i.e., does the sequence satisfy the LTL-formula $\Diamond\Box >$) or not?

– If so, then use Lemma 1 to reach that point, say $s' \sim t'$, then apply *sim_weaken* to deduce that $s' \simeq t'$, and use the *sim_wstep* to construct the infinite trace.
– If they are not, *tr* contains infinitely many transitions that are not in the $>$ relation ($\Box\Diamond \not>$ in LTL), and so using *sim_step*, we can produce an infinite trace for the source. □

5 Proofs of the Optimisations

This section gives brief outlines the formal Coq proofs of correctness for the three optimisations that were presented in §3.

Fence Elimination 1. We verify this optimisation by measured simulation.

Take $>$ to be empty relation (which is trivially well-founded) and $s \sim t$ the relation requiring that (i) the control-flow-graph of t is the optimised version of the CFG of s, (ii) s and t have identical program counters, local states, buffers and memory, and (iii), for each thread i, if the analysis for i's program counter returned $\perp$, then i's buffer is empty.

It is straightforward to show that each target step is matched exactly by the corresponding step of the source program. In the case of a `nop` instruction, this could arise either because of a `nop` in the source or because of a removed `fence`. In the latter case, the analysis will have returned $\bot$ and so, according to $\sim$, the thread's buffer is empty and so the `fence` proceeds (i.e., it does not block). Note that condition (*iii*) is straightforward to re-establish after each step, because the transfer function, T_1, returns $\bot$ only after a fence or an atomic instruction (when the buffer is necessarily empty) and $\top$ whenever something could have been added to the buffer (i.e., at `store` instruction or a function call).

Fence Elimination 2. We verify this optimisation by exhibiting a weaktau simulation, for which we shall need the following two auxiliary definitions:

- Define $s \equiv_i t$ to hold whenever thread i of s and t have identical program counters, local states and buffers.
- Define $s \rightsquigarrow_i s'$ if thread i of s can execute a sequence of `nop`, `op`, `store` and `fence` instructions and end in the state s'.

Take $s \sim t$ the relation requiring that (*i*) t's CFG is the optimised version of s's CFG, (*ii*) s and t have identical memories, (*iii*), for each thread i, either $s \equiv_i t$ or the analysis for i's program counter returned $\bot$ (meaning that there is a later fence in the CFG with no reads in between) and there exists a state s_0 such that $s \rightsquigarrow_i s_0$ and $s_0 \equiv_i t$.

Take $s \simeq t$ to be the relation requiring that: (*i*) the CFG of t is the optimised version of the CFG of s, and (*ii*), for each thread i, there exists s_0 such that $s \rightsquigarrow_i s_0$ and $s_0 \equiv_i t$. It is easy to see that $\sim$ and $\simeq$ satisfy *sim_weaken*: that is, for all s and t, $s \sim t$ implies $s \simeq t$.

Finally, let $t > t'$ be defined whenever $t \xrightarrow{\tau} t'$ by a thread executing a `nop`, an `op`, or a `store` instruction.

To prove *sim_step*, we match every step of the target with the corresponding step of the source whenever the analysis at the current program point of the thread doing the step returns $\top$. It is possible to do so, because by the simulation relation ($s \sim t$), we have $s \equiv_i t$.

Now, consider the case when the target thread i does a step and the analysis at the current program point returns $\bot$. According to the simulation relation ($\sim$), we have $s \rightsquigarrow_i s_0 \equiv_i t$. Because of the transfer function, T_2, that step cannot be a `load` or a `call`/`return`/`threadcreate`. We are left with the following cases:

- `nop` (either in the source program or because of a removed fence), `op`, or `store`. In these cases, we stutter in the source, i.e. do $s \sim t'$. This is possible because we can perform the corresponding transition from s_0 (i.e., there exists an s' such that $s \rightsquigarrow_i s_0 \rightsquigarrow_i s' \equiv_i t'$).
- `fence`, `atomic`: This is matched by doing the sequence of transitions from s to s_0 followed by flushing the local store buffer and finally executing the corresponding `fence` or `atomic` instruction from s_0.
- Thread i unbuffering: If i's buffer is non-empty in s, then unbuffering one element from s preserves the simulation relation. Otherwise, if i's buffer is

empty, then there exists an s' such that $s \rightsquigarrow_i s' \rightsquigarrow_i s_0$ and i's buffer in s' has exactly one element. Then the transition from $t \xrightarrow{\tau} t'$ is simulated by first doing $s \xrightarrow{\tau}^* \xrightarrow{\tau} s'$ followed by an unbuffering from s', which preserves the simulation relation.

To prove *wsim_step*, we simulate a target thread transition by doing the sequence of transitions from s to s_0 followed by executing the corresponding instruction from s_0.

Partial Redundancy Elimination. Even though this optimisation was the most complex to implement, its proof was actually the easiest. What this optimisation does is to replace some `nop` instructions by `fence` instructions depending on some non-trivial analysis. However, as far as correctness is concerned, it is always safe to insert a `fence` instruction irrespective of whatever analysis was used to used to decide to perform the insertion. Informally, this is because inserting a memory fence just restricts the set of behaviours of the program; it never adds any new behaviour.

In the formal proof, we take the simulation relation to be equality except on the programs themselves, where we require the target program to be the 'optimised' version of the source program. Executing the inserted `fence` instruction in the target is simulated by executing the corresponding `nop` in the source.

6 Coq Experience

Figure 6 presents the size of our development broken down in lines of extracted code, lines of specifications (i.e., definitions and statements of lemmata and theorems), and of proof script. Blank lines and comments are not counted. For comparison, the whole of CompCertTSO is roughly 85 thousand lines.

Line counts do not accurately reflect the time taken to carry out those proofs. The definitions of program traces, of the various kinds of simulations and their properties (namely, that they imply trace inclusion) took about a month. The main challenge was coming up with the definition of a weaktau simulation; the proof that weaktau simulations imply trace inclusion took us less than a day once we had come up with the definition. Prior to that we had spent two man-months formalizing backward (prophecy) simulations [17] and trying to use them

	Code	Specs	Proofs
Traces & simulations	–	490	358
Auxiliary memory lemmata	–	162	557
Fence elimination 1	68	213	319
Fence elimination 2	68	336	652
Fence introduction (PRE)	138	117	127
Total	274	1318	2013

Fig. 6. Size of formal development in lines as reported by `coqwc`

to verify our second fence elimination optimisation, albeit unsuccessfully. The trace inclusion proofs were moderately tricky to formalize in Coq because of coinductive reasoning and the use of the axiom of choice, for which we assumed the classical epsilon operator.

Coding up the fence elimination optimisations took half a day, and so did the soundness proof of the first one. Proving the correctness of the second optimisation required in total about three man-months of effort, reduced to less than a week once we defined the weaktau simulation, a significant part of which was devoted to developing generic infrastructure to reason about executions resulting in a memory error. Finally, PRE took a couple of days to implement and two hours to prove correct. In total, we spent about 5 man-months on this project.

7 Related Work

The problem of inserting memory barriers so that a program admits only SC executions has been, and still is, a central research topic since Sasha and Snir's seminal paper on delay set analysis [26]. Most studies of this problem [26,2,6] have mostly been in terms of hypothetical program executions and, unlike our work, have not been integrated in a working compiler.

There is also some compiler work. Lee and Padua [15] describe an algorithm based on dominators for inserting memory fences, while Sura et al. [27] focus on the more practical aspects, e.g., on how to approximate delay sets by performing cheaper whole-program analyses coupled with an escape analysis. While these works perform much more sophisticated analyses than the ones we implemented, unfortunately none of them comes with a mechanised soundness proof.

Another line of research [5,13,14] uses model checking techniques to insert fences to ensure SC. While these techniques may insert fewer fence instructions for small intricate concurrent libraries, they often guarantee soundness only for some clients of those libraries, and are too expensive to perform in a general-purpose compiler.

8 Conclusion

We have reported on the implementation of three barrier elimination optimisations within CompCertTSO and on their mechanised correctness proof in Coq. Our results suggest that reasoning about compiler optimisations for weak memory models are good candidates for mechanisation, and believe that this work will facilitate the formal study of more advanced compiler optimisations for concurrent programs within verified compilers.

Acknowledgements. We thank Mark Batty for discussion about fence optimisations, Suresh Jagannathan for pointing us at partial redundancy elimination, and Jaroslav Ševčík and Peter Sewell for comments on an early draft. We acknowledge funding from INRIA *équipes associées MM.*

References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. Theor. Comput. Sci., 253–284 (1991)
2. Alglave, J.: A shared memory poetics. Ph.D. thesis, Université Paris 7 (2010)
3. Becker, P.: Working draft, standard for programming language C++, n3090=10-0080 (March 2010)
4. Blazy, S., Leroy, X.: Mechanized semantics for the Clight subset of the C language. J. Autom. Reasoning 43(3), 263–288 (2009)
5. Burckhardt, S., Alur, R., Martin, M.M.K.: CheckFence: checking consistency of concurrent data types on relaxed memory models. In: PLDI (2007)
6. Burckhardt, S., Musuvathi, M., Singh, V.: Verifying local transformations on relaxed memory models. In: Gupta, R. (ed.) CC 2010. LNCS, vol. 6011, pp. 104–123. Springer, Heidelberg (2010)
7. Cao Minh, C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: IISWC (2008)
8. The Compcert verified compiler, v. 1.5 (August 2009), http://compcert.inria.fr/release/compcert-1.5.tgz
9. CompCertTSO (2011), http://www.cl.cam.ac.uk/~pes20/CompCertTSO
10. Dice, D., Shalev, O., Shavit, N.N.: Transactional locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
11. Eide, E., Regehr, J.: Volatiles are miscompiled, and what to do about it. In: EMSOFT (2008)
12. Fraser, K.: Practical Lock Freedom. Ph.D. thesis, University of Cambridge, also available as Tech. Report UCAM-CL-TR-639 (2003)
13. Huynh, T.Q., Roychoudhury, A.: Memory model sensitive bytecode verification. Form. Methods Syst. Des. 31, 281–305 (2007)
14. Kuperstein, M., Vechev, M., Yahav, E.: Automatic inference of memory fences. In: FMCAD (2010)
15. Lee, J., Padua, D.A.: Hiding relaxed memory consistency with a compiler. IEEE Trans. Comput. 50, 824–833 (2001)
16. Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning 43(4), 363–446 (2009), http://gallium.inria.fr/~xleroy/publi/compcert-backend.pdf
17. Lynch, N., Vaandrager, F.: Forward and backward simulations I: untimed systems. Inf. Comput. 121, 214–233 (1995)
18. Manson, J., Pugh, W., Adve, S.: The Java memory model. In: POPL (2005)
19. Morel, E., Renvoise, C.: Global optimization by suppression of partial redundancies. Commun. ACM 22, 96–103 (1979)
20. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 391–407. Springer, Heidelberg (2009)
21. Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D.: Understanding POWER multiprocessors. In: PLDI (2011)
22. Ševčík, J., Aspinall, D.: On validity of program transformations in the java memory model. In: Ryan, M. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 27–51. Springer, Heidelberg (2008)
23. Ševčik, J.: Safe optimisations for shared-memory concurrent programs. In: PLDI (2011)

24. Ševčik, J., Vafeiadis, V., Zappa Nardelli, F., Jagannathan, S., Sewell, P.: Relaxed-memory concurrency and verified compilation. In: POPL (2011)
25. Sewell, P., Sarkar, S., Owens, S., Zappa Nardelli, F., Myreen, M.O.: x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM 53(7), 89–97 (2010)
26. Shasha, D., Snir, M.: Efficient and correct execution of parallel programs that share memory. ACM Trans. Program. Lang. Syst. 10, 282–312 (1988)
27. Sura, Z., Fang, X., Wong, C.-L., Midkiff, S.P., Lee, J., Padua, D.: Compiler techniques for high performance sequentially consistent Java programs. In: PPoPP (2005)
28. Terekhov, A.: Brief tentative example x86 implementation for C/C++ memory model. cpp-threads mailing list (2008), `http://www.decadent.org.uk/pipermail/cpp-threads/cpp-threads/2008-December/001933.html`
29. Treiber, R.K.: Systems programming: Coping with parallelism. Tech. rep. (1986)

An Efficient Static Trace Simplification Technique for Debugging Concurrent Programs

Jeff Huang and Charles Zhang

Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
{smhuang,charlesz}@cse.ust.hk

Abstract. One of the major difficulties in debugging concurrent programs is that the programmer usually experiences frequent thread context switches, which perplexes the reasoning process. This problem can be alleviated by trace simplification techniques, which produce the same computation process but with much fewer number of context switches. The state of the art trace simplification technique takes a dynamic approach and does not scale well to large traces, hampering its practicality. We present a static trace simplification approach, SimTrace, that dramatically improves the efficiency of trace simplification through reasoning about the computation equivalence of traces offline. By constructing a dependence graph model of events, our trace simplification algorithm scales linearly to the trace size and quadratic to the number of nodes in the dependence graph. Underpinned by a trace equivalence theorem, we guarantee that the results generated by SimTrace are sound and no dynamic program re-execution is required to validate the trace equivalence. Our experiments show that SimTrace scales well to traces with more than 1M events, making it attractive to practical use.

Keywords: Trace simplification, Debugging, Concurrent program.

1 Introduction

Software is becoming increasingly concurrent due to the prevalence of multicore hardware. Unfortunately, the unique non-deterministic nature of concurrency makes debugging concurrent programs notoriously difficult. For instance, according to a recent report [9], the average bug fixing time of concurrency bugs (137 days) is 2.58 times longer than that of sequential ones in the MySQL[1] project. In our experience, the challenge of debugging concurrent programs comes from two main sources. First, concurrency bugs are hard to reproduce, as they may only manifest under certain specific thread interleavings. Due to the non-deterministic thread interleavings, a concurrency bug manifested in an earlier execution may "disappear" when the programmer attempts to reproduce it [17]. Second, concurrency bugs are hard to understand. A buggy concurrent program execution often contains many thread context switches. Since most programmers are used

[1] http://www.mysql.com/

E. Yahav (Ed.): SAS 2011, LNCS 6887, pp. 163–179, 2011.

to thinking sequentially, they have to jump frequently from the context of one thread to another for reasoning about a concurrent program execution trace. These frequent context switches significantly impair the efficiency of debugging concurrent programs [14].

Researchers have studied the first problem for several decades [5] and numerous effective approaches [12,16,20,19,13] are proposed for reproducing concurrency bugs. For the second problem, Jalbert and Sen [14] have recently proposed a *dynamic* trace simplification technique, Tinertia, for reducing the number of thread interleavings in a buggy execution trace. From a high level perspective, Tinertia iteratively transforms an input trace that satisfies a certain property to another trace satisfying the same property but with fewer thread context switches. Tinertia is valuable in improving the debugging efficiency of concurrent programs as it prolongs the sequential reasoning of concurrent program executions and reduces frequent "context switches". However, since Tinertia is a dynamic approach, it faces serious efficiency problems when used in practice. To reduce every single context switch, Tinertia has to re-execute the program at least once to validate the equivalence of the transformed trace. It is very hard for Tinertia to scale to large traces as the program re-execution typically requires controlling the thread scheduler to follow the scheduling decisions in the transformed trace, which is often 5x to 100x slower than the native execution [20]. From the time complexity point of view, the total running time of Tinertia is cubic to the trace size [14].

In this paper, we present a *static* trace simplification technique, SimTrace, that dramatically improves the efficiency of trace simplification through the offline reasoning of the computation equivalence of traces. The key idea of SimTrace is that we can statically guarantee the trace equivalence by leveraging the dependence relations between events in the trace. By presenting a formal modeling of dependence relation, we show a theorem of trace equivalence that any rescheduling of the events in the trace respecting the dependence relation is equivalent to the given trace. The trace equivalence is not limited to any specific property but general to all properties that can be defined over the program state. Underpinned by the trace equivalence theorem, SimTrace is able to perform the trace simplification completely offline, without any dynamic re-execution to validate the intermediate simplification result, which significantly improves the efficiency of the trace simplification.

In our analysis, we first build a *dependence graph* that encodes all the dependence relations between events in the trace. The dependence graph is a direct acyclic graph in which each node in the graph represents a corresponding event or event sequence by the same thread in the trace, and each edge represents a *happens-before* relation or a data dependence between two events or event sequences. The dependence graph is sound in that it encodes a complete set of dependence relations between the events. And the trace equivalence theorem guarantees that any topological sort of the dependence graph produces an equivalent trace to the original trace.

Taking the advantage of the dependence graph, we reduce the trace simplification problem to a *graph merging problem*, of which the objective is minimizing the size of the graph. To address this problem, we developed a graph merging algorithm that performs a sequence of merging operations on the graph. Each merging operation is applied on two consecutive nodes by the same thread in the graph, and it consolidates the two nodes if a merging condition is satisfied. The merging condition is that the edge connecting the two merged nodes is the only path connecting them in the graph, which can be efficiently checked by computing the reachability relation between the two nodes.

Finally, SimTrace performs a topological sort on the reduced dependence graph and generates the simplified trace. The total running time of SimTrace is linear in the size of the trace and quadratic in the number of the nodes in the initial dependence graph. SimTrace is very efficient in practice, since the size of the initial dependence graph is often much smaller than that of the original trace. Moreover, guaranteed by the sound dependence graph model, SimTrace is completely offline and does not require any re-execution of the program for validating the equivalence of the simplified trace.

For the general trace simplification problem of generating equivalent traces with *minimum* context switches, Jalbert and Sen [14] have proved that this problem is NP-hard. Like Tinertia, SimTrace does not guarantee the globally optimal simplification but a local optimum. However, our evaluation results using a set of multithreaded programs show that SimTrace has good performance that is able to significantly reduce the context switches in the trace. For instance, for the input trace of the *Cache4j* subject with 1,225,167 events, SimTrace is able to reduce the number of context switches from 417 to 33 (with 92% reduction percentage) in 592 seconds. The overall reduction percentage of SimTrace ranges from 65% to 97% in our experiments.

Being an offline analysis technique, SimTrace is complementary to Tinertia. For the sake of efficiency, our modeling of the dependence relation does not consider the runtime value dependencies between events in the trace and hence may be too strict in preventing further trace simplification. As Tinertia utilizes the runtime verification regardless of the dependence relation, it might be able to explore more simplification opportunities that are beyond the strict dependence relation. A good match between SimTrace and Tinertia for the trace simplification is to apply SimTrace as a front-end and use Tinertia as a back end. By working together, we can achieve both the trace simplification efficiency and effectiveness at the same time, i.e., to more efficiently generate a simplified trace with fewer context switches.

To sum up, the key contributions of this paper are as follows:

– We present an efficient static trace simplification technique for reducing the number of thread context switches in the trace.
– We show a theorem of trace equivalence that is general to all properties defined over the program state. This theorem provides the correctness guarantee of the static trace simplification without any dynamic program re-execution to validate the intermediate simplification result.

- We present a sound graph modeling of the dependence relation between events in the trace, which allows us to develop efficient graph merging algorithms for the trace simplification problem.
- We evaluate our approach on a number of multithreaded applications and the results demonstrate the efficiency and the effectiveness of our approach.

The rest of the paper is organized as follows: Section 2 describes the problem of trace simplification; Section 3 presents our algorithm; Section 4 reports our evaluation results; Section 5 discusses the related work and Section 6 concludes this paper.

2 Preliminaries

In this section, we first describe a simple but general concurrent program execution model. Based on this model, we then formally introduce the general trace simplification problem.

2.1 Concurrent Program Execution Model

To formally present the trace simplification problem and to prove the trace equivalence theorem, we need to define a concurrent program execution model with precise execution semantics. Previous work [8,7,24] has described the concurrent program execution models in several different ways for different analysis purposes. To make our approach general, we define a model in a similar style to [8].

A concurrent program in our language consists of a set of concurrently executing threads $\mathbb{T} = \{t_1, t_2, ...\}$ that communicate through a global store σ. The global store consists of a set of variables $\mathbb{S} = \{s_1, s_2, ...\}$ that are shared among threads. Each thread has also its own local store π, consisting of the local variables and the program counter to the thread. We use $\sigma[s]$ to denote the value of the shared variable s on the global store. Each thread executes by performing a sequence of actions on the global store or the thread's own local store. Let α refer to an action and $var(\alpha)$ the variable accessed by α. If $var(\alpha)$ is a shared variable, we call α a *global action*, otherwise it is a *local action*. Note that for any global action, it operates on only *one variable* on the global store. This is also true for synchronization actions, though they are only enabled when certain pre-conditions are met. For local actions, the number of accessed variables on the local store is not important in our modeling. We next explain the execution semantics.

The program execution is modeled as a sequence of transitions defined over the program state $\Sigma = (\sigma, \Pi)$, where σ is the global store and Π is a mapping from thread identifiers t_i to the local store π_i of each thread. Since the program counter is included in the local store, each thread is deterministic and the next action of t_i is determined by t_i's current local store π_i. Let α_k be the k^{th} action in the global order of the program execution and Σ^{k-1} be the program state just before α_k is performed (Σ^0 is the initial state), the state transition sequence is:

$$\Sigma^0 \xrightarrow{\alpha_1} \Sigma^1 \xrightarrow{\alpha_2} \Sigma^2 \xrightarrow{\alpha_3} \ldots$$

Given a concurrent system described above, the execution semantics of actions are defined as follows:

Local action When a local action is performed by a thread, only the local store of that thread is changed to a new state determined by its current state. The global store and the local stores of the other threads remain the same.

Global action When a global action is performed by a thread t_i on the shared variable s, only s and π_i are changed to new states. The states of all the other shared variables on the global store as well as the local stores of all the other threads remain the same. To make the execution model general to different programming languages, we consider the following types of global actions:

- READ - a thread reads the value of a shared variable in the global store into its local store;
- WRITE - a thread assigns some value to a shared variable in the global store;
- LOCK - a thread acquires a lock;
- UNLOCK - a thread releases a lock;
- FORK - a thread forks a new thread;
- JOIN - a thread joins the termination of another thread;
- START - a *dummy* action indicating that a thread is ready to run;
- EXIT - the last action of a thread;
- SIGNAL - a thread sets the value of a conditional variable to 1;
- WAIT - a thread waits for a conditional variable to become 1 and resets it back to 0 after it becomes 1.

2.2 General Trace Simplification Problem

Definition 1. *A* **trace** *is the action sequence $\langle \alpha_k \rangle$ of a program execution.*

Definition 2. *A* **context switch** *occurs when two consecutive actions in the trace are performed by different threads.*

Let $\Gamma(\alpha)$ denotes the owner thread of action α. Let δ denote a trace containing N actions and $\delta[k]$ the k^{th} action in δ, and let $CS(\delta)$ denote the number of context switches in δ, we have $CS(\delta) = \Sigma_{k=1}^{N-1} u_k$ where u_k is a binary variable s.t. $u_k = 1$ if $\Gamma(\delta[k]) \neq \Gamma(\delta[k+1])$ and $u_k = 0$ otherwise.

Definition 3. *Two traces are* **equivalent** *if they drive the same initial program state to the same final program state.*

Given a trace as the input, the general trace simplification problem is to produce an output trace that is *equivalent* to the input trace and has minimum number of context switches among all the equivalent traces. To state more formally, suppose an input trace δ drives the program state to Σ^N, the general trace simplification problem is: *given δ, output a δ' s.t. $\Sigma^N = \Sigma'^N$ and $CS(\delta')$ is minimized.* Notice that the program state here is not limited to any local store or the global store but includes both the global store and the local stores of all the threads. In other words, the trace simplification problem defined above is general to all properties defined over the program state.

The basic idea for reducing the context switches in a trace is to reschedule the actions in the trace such that more actions by the same thread are placed next to each other. A naïve approach is to exhaustively generate all permutations of the events in the trace and pick an equivalent one with the smallest number of context switches. However, this naïve approach requires checking N! permutations which is highly inefficient. A better approach is to repeatedly move the interleaving actions to some non-interleaving positions and then consolidate the neighboring actions by the same thread. However, there are two major challenges in this approach. First, how to ensure the rescheduled trace is feasible and also equivalent to the input trace? Second, how to make sure the output trace is optimal, i.e., has the minimum number of context switches among all the equivalent traces?

3 SimTrace: Efficient Static Trace Simplification

We address the trace simplification problem by leveraging the dependence relationship between actions in the trace. For the first challenge, we show that the trace equivalence can be guaranteed by respecting the dependence relation during the rescheduling process. For the second challenge, since Jalbert and Sen [14] have proved it is *NP-hard*, we present an efficient algorithm, SimTrace, that guarantees to generate a locally optimal solution. In this section, we first describe our modeling of the dependence relation. Based on the modeling, we describe a theorem of trace equivalence and offer a detailed proof. After that, we present the full SimTrace algorithm.

3.1 Modeling of the Dependence Relation

Previous work has proposed many causal models [21,3,15,18,26] that characterize the dependence relationship between actions in the trace. Among them, most models are developed for checking concurrency properties such as data race and atomicity violations, and they are tailored for the specific property. Different from these models, as we are dealing with all properties over the program state, we have to consider a general model that works for all properties.

Moreover, to support efficient trace simplification, the model should be as simple as possible. Although using a more precise dependence relation model, such as the maximal causal model [21] or the guarded independence model [26], may give us more freedom to simplify the trace (which can further reduce the context switches), such a model is often very expensive to construct in practice, because it requires to track all the value dependencies between events along the program branches and all the correlated control flow decisions in the program execution. We thus use a strict model defined as follows:

Definition 4. *The* **dependence relation** *($\rightarrow$) for a trace δ is the smallest transitive closure over the actions in δ, such that $a_i \rightarrow a_j$ holds whenever a_i occurs before a_j and one of the following holds:*

- **Local dependence relation**
  - *Program order* - a_i immediately precedes α_j in the same thread.
- **Remote dependence relation**
  - *Synchronization order* - a_i and a_j are consecutive synchronization actions by different threads on the same shared variable. There are four types of synchronization orders:
    - ∗ UNLOCK→LOCK: a_i is the UNLOCK action that releases the lock acquired by the LOCK action a_j;
    - ∗ FORK→START: a_i is the FORK action that forks the thread whose START action is a_j;
    - ∗ EXIT→JOIN: a_i is the EXIT action of a thread that the JOIN action a_j joins;
    - ∗ NOTIFY→WAIT: a_i is the NOTIFY action that sets the conditional variable the WAIT action a_j waits for;
  - *Conflicting order* - a_i and a_j are consecutive conflicting actions by different threads on the shared variable. There are three types of conflicting orders:
    - ∗ WRITE→READ: a_i is a WRITE action and a_j is a READ action;
    - ∗ READ→WRITE: a_i is a READ action and a_j is a WRITE action;
    - ∗ WRITE→WRITE: both a_i and a_j are WRITE actions.

Given a dependence relation $a_i \rightarrow a_j$, If a_i and a_j are from different threads, we say a_i has a remote outgoing dependence to a_j, and similarly, a_j has a remote incoming dependence to a_i; Otherwise, we say a_i has a local outgoing dependence to a_j and a_j has a local incoming dependence to a_i.

It is important to notice that the remote dependence relations in our model are all between actions accessing the same shared variable. Therefore, context switches between threads accessing different variables in the trace are allowed to be reduced in our model. Nevertheless, also note that the remote dependence in our modeling includes both the synchronization order and the conflicting order, which may actually be unnecessary if we consider the value dependencies between events. For example, two writes to the same variable with the same value can be permuted without affecting the correctness of the trace. The main purpose for using this strict model is that we want to efficiently and safely guarantee the trace equivalence, towards which we must ensure that every action in the simplified trace reads or writes the same value as its corresponding action in the original trace.

3.2 A Theorem of Trace Equivalence

Based on our model of the dependence relation in Section 3.1, we have the following theorem of trace equivalence:

Theorem 1. *Any rescheduling of the actions in a trace respecting the dependence relation defined in Definition 4 generates an equivalent trace.*

Proof. (Sketch) Let δ denote the input trace with size N and δ' an arbitrary rescheduling of δ respecting the dependence relation, and suppose δ and δ' drive the program state from the same initial state Σ^0 and Σ'^0 to Σ^N and Σ'^N, respectively. Our goal is to prove $\Sigma'^N = \Sigma^N$. The main insight of the proof is that, by respecting the order defined by the dependence relation, every action in the rescheduled trace reads or writes the same value on the program state as its corresponding action in the input trace, and hence the rescheduled trace drives the program to the same final state as that of the input trace. We provide the full detailed proof in the Appendix A. Readers may skip it at this moment. ■

Note that Theorem 1 is related to but different from the equivalence axiom of the Mazurkiewicz traces [1] in the trace theory, which provides an abstract model of reasoning about trace equivalence based on the partial order relation between events. We prove Theorem 1 in the context of concurrent program execution based on the concrete modeling of the action semantics and the computation effect in the trace.

In spite of the intuitiveness, Theorem 1 forms the basis of static trace simplification as it guarantees every rescheduling of the actions in the trace that respects the dependence relation produces a valid simplification result, without the need of any runtime verification. In other words, as long as we do not violate the order defined by the dependence relation, we can safely reschedule the events in the trace without worrying about correctness of the final result.

3.3 SimTrace Algorithm

Our algorithm starts by constructing from the input trace a dependence graph (see Definition 5), which encodes all the actions in the trace as well as the dependence relations between the actions. We then simplify the dependence graph by ordinally performing a "merging" operation on two consecutive nodes by the same thread in the graph. When the dependence graph cannot be further simplified, our approach applies a simple topological sort on the graph to produce the final simplified trace.

Definition 5. *A* **dependence graph** $G = (V, E)$*, built upon a trace, is a directed acyclic graph in which each* $v \in V$ *corresponds to a sequence of consecutive actions by the same thread started by a unique action that has remote incoming dependence. For each edge, there is a labeling relation* $L : E \rightarrow \{local, remote\}$ *such that each local edge connects neighboring nodes by the same thread, and each remote edge connects nodes by different threads meaning that there are dependence relations from some actions in one node to some actions in the other node.*

Note that the dependence graph is directed acyclic graph. Otherwise it indicates there are cyclic dependences between events in the trace, which is impossible according to our dependence relation model. We next describe our algorithms for constructing and simplifying the dependence graph in detail.

Algorithm 1. ConstructDependenceGraph(δ)

```
input: $\delta$ (a trace)
output: $graph$ (the dependence graph built from $\delta$)
$map_{t2n} \leftarrow$ empty map from a thread identifier to its current graph node
$t_{old} \leftarrow$ null
for $i \leftarrow 0$ to $|\delta|$-1 do
    $t_{cur} \leftarrow$ the thread identifier of the action $\delta[i]$
    $node_{cur} \leftarrow map_{t2n}(t_{cur})$
    if $node_{cur}$ is null then
        $node_{cur} \leftarrow$ new node($\delta[i]$)
        $map_{t2n}(t_{cur}) \leftarrow node_{cur}$
        add node $node_{cur}$ to $graph$
    else
        if $\delta[i]$ has remote incoming dependence and $t_{cur} \neq t_{old}$ then
            $node_{old} \leftarrow node_{cur}$
            $node_{cur} \leftarrow$ new node($\delta[i]$)
            add node $node_{cur}$ to $graph$
            add local edge $node_{old} \dashrightarrow node_{cur}$ to $graph$
            for each action $a$ with remote outgoing dependence to $\delta[i]$ do
                $node_a \leftarrow$ the node to which $a$ belongs
                add remote edge $node_a \rightarrow node_{cur}$ to $graph$
            end for
        else
            add action $\delta[i]$ to $node_{cur}$
        end if
    end if
    $t_{old} \leftarrow t_{cur}$
end for
```

Dependence Graph Construction. Algorithm 1 shows our algorithm for constructing the dependence graph. Given an input trace, we first conduct a linear scan of all the actions in the trace to build the smallest dependence relation between actions, according to our model in Section 3.1. We then visit each action in their appearing order in the trace once to construct the dependence graph according to Definition 5. Our construction of the dependence graph leverages the observation that most of the dependence relations in the trace are local dependencies within the same thread, while the number of remote dependence relations are comparatively much smaller. We can hence greatly reduce the size of the initial dependence graph by shrinking consecutive actions with only local dependence between them into a single node. The running time of Algorithm 1 is linear to the trace size.

Note that, in our dependence graph construction process, each node in the initial dependence graph has exactly two incoming edges except the root node: a local incoming edge and a remote incoming edge. The number of edges in the graph is thus less than twice the number of nodes in the graph. Moreover, since each node in the dependence graph may represent a sequence of actions in the trace, the number of nodes in the graph is much smaller than the original trace size. As a result, performing a topological sort on the dependence graph is much more efficient than that on the original trace.

Simplifying Dependence Graph. Following Theorem 1, it is easy to see that any topological sort of the initial dependence graph produces a correct answer to our problem, i.e., generates an equivalent trace to the input trace. However, to make the resultant trace as simple as possible, i.e., to minimize the context switches, we have to wisely choose the next node in each sorting step during the topological sort, which is a difficult problem with no existing solution or even good approximation algorithm, to our best knowledge.

We formulate this problem as an optimization problem on the number of nodes in the dependence graph and use a graph merging algorithm to compute a locally optimal solution to it. Before describing the formulation, let us first introduce a dual notion of context switch:

Definition 6. *A* **context continuation** *occurs when two consecutive actions in the trace are performed by the same thread.*

Let $CC(\delta)$ denote the number of context continuations in a trace δ, we have the following lemma:

Lemma 1. *Minimizing $CS(\delta)$ is equivalent to maximizing $CC(\delta)$.*

Proof. Traversing the trace once, it is easy to see that for each action, either $CS(\delta)$ or $CC(\delta)$ is incremented. Thus, $CS(\delta) + CC(\delta) = N - 1$. Hence, $CS(\delta)$ is minimized when $CC(\delta)$ is maximized.

Therefore, our goal becomes to maximize the number of context continuations in the simplified trace. Now let us consider the action sequence represented by each node in the dependence graph. Since all actions in the same action sequence are performed by the same thread, their number of context continuations are already optimized. The remaining possible context continuations can only come from actions that are in different action sequences. Mapping this back to the dependence graph and because nodes representing action sequences by the same thread are connected by local edges, we have the following lemma:

Lemma 2. *Minimizing $CS(\delta)$ is equivalent to maximizing the number of context continuations contributed by local edges in the dependence graph.*

Consider a local edge in the graph, if the action sequences represented by the two nodes connected by this local edge are consolidated together, it will contribute one context continuation. Let us call a *merging* operation as the consolidating of two nodes connected by a local edge in the dependence graph. As each merging operation eliminates a local edge and correspondingly reduces one node in the dependence graph, it is easy for us to get the following theorem:

Theorem 2. *Minimizing $CS(\delta)$ is equivalent to minimizing the number of nodes in the dependence graph.*

Following Theorem 2, our objective is performing as many merging operation as possible so as to minimize the number of nodes in the dependence graph. However, recall that the dependence relation between actions in the trace must be respected. Therefore, we cannot arbitrarily perform the merging operation without satistifying a certain pre-condition: the merging condition is that the to-be-merged two nodes are connected by the local edge only. Otherwise, the resultant graph after the merging operation would become cyclic that violates the definition of dependence graph. Mapping this back to the semantics of the dependence relation, the merging condition simply requires that there should not exist another dependent action in the trace that interleaves the two action sequences represented by the to-be-merged two nodes in the dependence graph. Checking the merging condition is simple because it only requires testing the reachability relation between the two merged nodes, which costs a linear running time in the number of nodes in the dependence graph using a simple algorithm[2].

Therefore, our dependence graph simplification algorithm (Algorithm 2) traverses each local edge in the dependence graph, and performs the merging operation if the merging condition is satisfied. This algorithm evaluates each local edge in the initial dependence graph once and each evaluation computes the reachability relation between two nodes once. The worst case time complexity is thus quadratic in the number of nodes in the initial dependence graph.

Algorithm 2. SimplifyDependenceGraph(*graph*)

```
input: $graph$ (the dependence graph)
output: $graph'$ (the simplified dependence graph)
$graph' \leftarrow graph$
for each local edge $node_a \rightarrow node_b$ in a random order do
    if $node_b$ is not reachable from $node_a$ except from the local edge then
        $merge(node_a, node_b, graph')$
    end if
end for
```

Notice that in our merging algorithm, the evaluation order of the local edges may affect the simplification result. Our algorithm does not guarantee a global optimum but produces a locally optimal simplification given the chosen evaluation order. To illustrate this problem, let us take the (incomplete) dependence graph in Figure 1 as an example. The graph contains 6 nodes, 3 local edges (denoted by dashed arrows ⇢), and 4 remote edge (denoted by solid arrows →): $a_1 \dashrightarrow a_2$, $b_1 \dashrightarrow b_2$, $c_1 \dashrightarrow c_2$, $a_1 \rightarrow b_2$, $c_1 \rightarrow b_2$, $b_1 \rightarrow a_2$ and $b_1 \rightarrow c_2$. If b_1 and b_2 are merged first, as shown in Figure 1 (a), it would produce the trace <a_1-c_1-b_1-b_2-c_2-a_2> that contains 4 context switches. However, the optimal solution is to merge a_1 and a_2, and c_1 and c_2, which produces the trace <b_1-a_1-a_2-c_1-c_2-b_2> that contains only 3 context switches. In fact, this problem is NP-hard (proved by Jalbert and Sen [14]), and there does not seem to exist an

[2] Theoretically, constant time graph reachability computation algorithms also exist. Please refer to [27] for details.

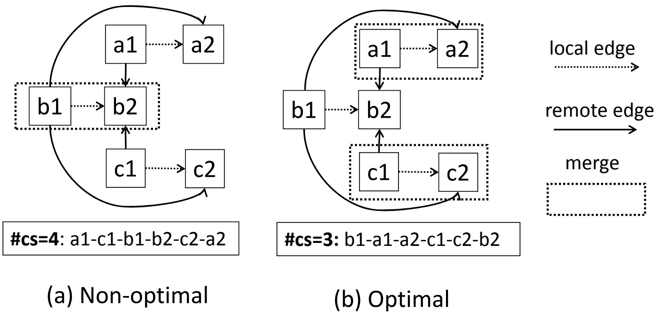


Fig. 1. A greedy merge may produce non-optimal result in (a). Unfortunately, the problem of producing the optimal result in (b) is NP-hard.

efficient algorithm for generating an optimal solution. Our algorithm thus picks a random order (or any arbitrary order) for evaluating the local edges. Though it does not guarantee to produce a global optimum, it is easy to see that our algorithm always produces a local optimum specific to the chosen evaluation order. That is, given the evaluation order of the local edges, our algorithm produces a trace with the fewest thread context switches.

4 Implementation and Experiments

We have implemented SimTrace as a prototype tool on top of our LEAP [13] record and replay framework for multithreaded Java programs. From the user's perspective, our tool consists of three phases. It first obtains a trace of a buggy concurrent Java program execution, which contains all the shared memory reads and writes as well as synchronization operations performed by each thread in the program. Then our tool applies the SimTrace algorithm on the trace and produces a simplified trace. In the third phase, it uses a replay engine to re-execute the program according to the scheduling decisions in the simplified trace. Our replayer is transparent to the programmers such that they can deterministically investigate the simplified buggy trace in a normal debugging environment.

The goal of our experiments is to investigated whether our approach is effective and how efficient it is in reducing the thread context switches in the trace. We chose eight widely used multithreaded Java benchmarks as the evaluation subjects (shown in the first column in Table 4). Each subject has one or more known concurrency bugs. *Philosopher* is a simulation of the classical dinning philosophers problem, *Bubble* is a buggy multithreaded version of the bubble sort algorithm, *TSP* is a multithreaded implementation of a parallel branch and bound algorithm for the travelling salesman problem, *StringBuffer* is an open library from Suns JDK 1.4.2, *Cache4j* is a thread-safe implementation of cache for Java objects with an atomicity violation bug, *Weblech* is a multi-threaded web site download and mirror tool, *SpecJMS* is SPEC's benchmark for evaluating the performance of enterprise message-oriented middleware servers based on JMS, and *Jigsaw* is W3C's leading-edge web server platform.

Similar to Tinertia [14], we use the random testing approach to generate the initial buggy trace for each subject. For each trace, we ran SimTrace multiple times with different evaluation orders of the local edges during our graph merging process (Algorithm 2). To remove the non-determinism related to random numbers, we fix the seed of random numbers to a constant in all the subjects. All experiments were conducted on a HP EliteBook running Windows 7 with 2.53GHz Intel Core 2 Duo processor and 4GB memory. For others to verify our experimental results, we put our implementation and all the evaluation subjects publicly available at `http://www.cse.ust.hk/prism/simtrace`.

Table 1. Experimental results. Data are averaged over 50 runs for each subject.

Program	LOC	Thread	SV	Trace Size	Time	Old Ctxt	New Ctxt	Reduction
Philosopher	81	6	1	131	6ms	51	18	65%
Bubble	417	26	25	1,493	23ms	454	163	71%
Elevator	514	4	13	2104	8ms	80	14	83%
TSP	709	5	234	636,499	149s	9272	1,337	86%
Cache4j	3,897	4	5	1,225,167	592s	417	33	92%
Weblench	35,175	3	26	11,630	57ms	156	24	85%
OpenJMS	154,563	32	365	376,187	38s	96,643	11,402	88%
Jigsaw	381,348	10	126	19,074	130ms	2396	65	97%

Table 4 shows the experimental results. All data are averaged over 50 runs. The first five columns show the statistics of the test cases, including the program name, the size of the program in lines of source code, the number of threads, the number of real shared memory locations that contain both read and write accesses from different threads in the given trace, and the length of the trace. The next four columns shows the statistics of our trace simplification algorithm (all on average), including the running time of our offline analysis, the number of context switches in the original trace, the number of context switches in the simplified trace and the percentage of reduction due to our simplification. The results show that our approach is promising in terms of both the trace simplification efficiency and the effectiveness. For the eight subjects, our approach is able to reduce the number of context switches in the trace by 65% to 97% on average. This reduction percentage is close to that of Tinertia, the reduction percentage of which ranges from to 32.1% to 97.0% in their experiments. More importantly, our approach is able to scale to much larger traces compared to that of Tinertia. For a trace with only 1505 events (which is the largest trace reported by Tinertia in their experiments), Tinertia requires a total of 769.3s to finish the simplification, while our approach can analyze a trace (the *Cache4j* subject) with more than 1M events within 600s. For a trace (the *Bubble* subject) with 1,493 events, our approach requires only 23ms to simplify it. Although a direct comparison between Tinertia and our approach is not applicable as the two approaches are implemented for different program languages (Tinertia is implemented for C/C++ programs) and have different evaluation subjects, we believe the statistical data provides some evidence demonstrating the value of our approach compared to the state of the art.

5 Related Work

From the perspective of the trace theory [6], our dependence graph modeling of traces is an instantiation of Mazurkiewicz traces [1], which models a class of equivalent traces with respect to an alphabet and a dependence relation over the alphabet. Different from the equivalence axiom of Mazurkiewicz traces [1], our trace equivalence theorem is built on top of the computation equivalence of program state transitions and relies on a concrete model of the program execution semantics.

Our model of the dependence relation is closely related to, but different from the classical happens-before model [15] and various other causal models [3,18,21,26], which do not enforce the conflicting orders used in our model, but rely on the value dependencies between the events to obtain a more precise causal model than ours. For example, Wang et al. [26] introduced a notion of guarded independence to enable precisely checking of atomicity violations, Chen et al. [3] proposed a notion of sliced causality that significantly reduces the size of the computed causality relation by slicing the causal dependences between the events, and Şerbănuţă et al. [21] proposed a maximal causal model that comprises in theory all consistent executions that can be derived from a given trace. Although using a more precise dependence relation model may further reduce the context switches in the trace, as explained in Section 2.1, our strict modeling enables us to more efficiently perform the trace simplification.

Besides the problem of reducing the context switches in the trace, another important problem for the trace simplification is to reduce the size of trace. Along this direction, Tallam et al. [23] proposed an execution reduction (ER) system for removing the events that are irrelevant to the bug property in the trace. Similar to SimTrace, it also builds a dependence graph based on the dynamic dependencies between events in the trace. The main difference is that the ER system in [23] lacks a formal modeling of the trace elements. Without formalizing the semantics of each event in the trace, it is not obvious to understand the characteristics and the correctness of the resultant dependence graph.

To help locate the cause of the bug in an error trace, a number of software model checking algorithms [11,10,2] have been proposed to minimize an error trace and extract useful counterexamples when a bug is found. Delta Debugging [28] has also been extended to discover the minimal difference in thread scheduling necessary to produce a concurrency-based error [4]. The difference between our approach and this school of techniques is that our approach is based on reduction on a single trace, while they in spirit are based on comparing related executions, e.g., similar pairs of executions such that one of them satisfies the specification and the other does not.

To help with automatic concurrent program verification, Sinha and Wang [22] recently proposed a novel approach that separates intra- and inter-thread reasoning by symbolically encoding the shared memory accesses and composing the intra-thread summaries based on a sequential consistent memory model. Since this approach avoids redundant bi-modal reasoning, it has been shown to greatly improve the efficiency of trace analysis over previous approaches [25].

6 Conclusion

We present an efficient static trace simplification technique for reducing the context switches in a concurrent program execution trace. By constructing a dependence graph model of events, our algorithm scales linearly to the trace size and quadratic to the number of nodes in the dependence graph. Moreover, underpinned by a trace equivalence theorem, our approach guarantees to generate an equivalent simplified trace without any dynamic program re-execution, making it attractive to practical use.

Acknowledgement. We thank the anonymous reviewers for their valuable feedbacks. This research is supported by RGC GRF grants 622208 and 622909.

References

1. Mazurkiewicz, A.: Trace theory. In: Rozenberg, G. (ed.) APN 1987. LNCS, vol. 266, Springer, Heidelberg (1987)
2. Ball, T., Naik, M., Rajamani, S.K.: From symptom to cause: localizing errors in counterexample traces. In: POPL (2003)
3. Chen, F., Roşu, G.: Parametric and sliced causality. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 240–253. Springer, Heidelberg (2007)
4. Choi, J.-D., Zeller, A.: Isolating failure-inducing thread schedules. In: ISSTA (2002)
5. Curtis, R., Wittie, L.D.: Bugnet: A debugging system for parallel programming environments. In: ICDCS (1982)
6. Diekert, V., Rozenberg, G.: The book of traces (1995)
7. Farzan, A., Madhusudan, P., Sorrentino, F.: Meta-analysis for Atomicity Violations under Nested Locking. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 248–262. Springer, Heidelberg (2009)
8. Flanagan, C., Freund, S.N.: Atomizer: a dynamic atomicity checker for multithreaded programs. In: POPL (2004)
9. Fonseca, P., Li, C., Singhal, V., Rodrigues, R.: A study of the internal and external effects of concurrency bugs. In: DSN (2010)
10. Groce, A., Chaki, S., Kroening, D., Strichman, O.: Error explanation with distance metrics. Int. J. Softw. Tools Technol. Transf. (2006)
11. Groce, A., Visser, W.: What went wrong: Explaining counterexamples. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 121–135. Springer, Heidelberg (2003)
12. Hower, D.R., Hill, M.D.: Rerun: Exploiting episodes for lightweight memory race recording. In: ISCA (2008)
13. Huang, J., Liu, P., Zhang, C.: LEAP: Lightweight deterministic multi-processor replay of concurrent Java programs. In: Hong, S., Iwata, T. (eds.) FSE 2010. LNCS, vol. 6147, Springer, Heidelberg (2010)
14. Jalbert, N., Sen, K.: A trace simplification technique for effective debugging of concurrent programs. In: FSE (2010)

15. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. CACM (1978)
16. Montesinos, P., Ceze, L., Torrellas, J.: Delorean: Recording and deterministically replaying shared-memory multi-processor execution efficiently. In: ISCA (2008)
17. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing heisenbugs in concurrent programs. In: OSDI (2008)
18. O'Callahan, R., Choi, J.-D.: Hybrid dynamic data race detection. In: PPoPP (2003)
19. Olszewski, M., Ansel, J., Amarasinghe, S.: Kendo: efficient deterministic multithreading in software. In: ASPLOS (2009)
20. Park, S., Zhou, Y., Xiong, W., Yin, Z., Kaushik, R., Lee, K.H., Lu, S.: PRES: probabilistic replay with execution sketching on multi-processors. In: SOSP (2009)
21. Şerbănuţă, T.F., Chen, F., Roşu, G.: Maximal causal models for sequentially consistent multithreaded systems. Technical report, University of Illinois (2010)
22. Sinha, N., Wang, C.: Staged concurrent program analysis. In: FSE (2010)
23. Tallam, S., Tian, C., Gupta, R., Zhang, X.: Enabling tracing of long-running multithreaded programs via dynamic execution reduction. In: ISSTA (2007)
24. Vineet, K., Chao, W.: Universal causality graphs: A precise happens-before model for detecting bugs in concurrent programs. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 434–449. Springer, Heidelberg (2010)
25. Wang, C., Chaudhuri, S., Gupta, A., Yang, Y.: Symbolic pruning of concurrent program executions. In: ESEC/SIGSOFT FSE (2009)
26. Wang, C., Limaye, R., Ganai, M., Gupta, A.: Trace-based symbolic analysis for atomicity violations. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 328–342. Springer, Heidelberg (2010)
27. Wang, H., He, H., Yang, J., Yu, P.S., Yu, J.X.: Dual labeling: Answering graph reachability queries in constant time. In: ICDE (2006)
28. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Trans. Softw. Eng. (2002)

Appendix: A Proof of Theorem 1

Proof. Let us say two actions are *equal iff* they perform the same operation on the same variable and also read and write the same value. The core of the proof is to prove the following lemma:

Lemma 3. *For any action α' in δ', suppose it is the n^{th} action of thread t_i, then α' is equal to the n^{th} action of t_i in δ.*

If Lemma 3 holds, we can prove Theorem 1 by applying it to the last actions that write to each variable in both δ and δ'. To prove Lemma 3, we first define a notion of *version number* and show two lemmas related to it:

Definition 7. *Every variable is associated with a* **version number** *such that it is (1) initialized to be 0 and (2) incremented by 1 when the variable is written by an action.*

Lemma 4. *For any action α' in δ', suppose it is the k^{th} action that writes to a variable s, then α' is also the k^{th} action that writes to s in δ.*

Proof. To prove Lemma 4, we only need to make sure the order of write actions on each variable is unchanged during the rescheduling of the trace from δ to δ'.

This follows our modeling of the dependence relation includes all synchronization orders and the WRITE→WRITE orders on the same variable. ■

Lemma 5. *For any action α' in δ', suppose it reads the variable s with version number p, then α' also reads s with the same version number p in δ.*

Proof. Similar to the proof of Lemma 4, since our model of the dependence relation includes all the synchronization orders and the WRITE→READ and READ→WRITE orders on the same variable, we guarantee every READ action in the rescheduled trace reads the value written by the same WRITE action as that in the original trace. ■

Let $\sigma[s]^p$ denote the value of variable s with version number p, we next prove Lemma 3 by deduction on the version number of each variable:

Proof. Consider the jth actions performed by t_i, denoted by $\alpha_{i:j}$ and $\alpha'_{i:j}$ in δ and δ' respectively. To prove $\alpha'_{i:j}$ is equal to $\alpha_{i:j}$, we need to satisfy two conditions. First, their actions should be the same, i.e., they perform the same operation on the same variable. Second, suppose they both operate on the variable s (which should be true if the first condition holds), the values of s before $\alpha'_{i:j}$ is performed in δ' should be the same as that in δ before $\alpha_{i:j}$ is performed. Let $\pi_{i:j}$ and $\pi'_{i:j}$ denote the local store of t_i after $\alpha_{i:j}$ is performed in δ and after $\alpha'_{i:j}$ is performed in δ', respectively. For the first condition, since the execution semantics determine that the next action of any thread is determined by that thread's current local store, we need to ensure (I) $\pi'_{i:j-1} = \pi_{i:j-1}$. For the second condition, suppose $\alpha_{i:j}$ and $\alpha'_{i:j}$ operate on s with version number p and p', respectively, we need to ensure (II) $\sigma'[s]^{p'} = \sigma[s]^p$.

Let's first assume Condition I holds, we prove $p' = p$ in Condition II. If $\alpha'_{i:j}$ writes to s, i.e., $\alpha'_{i:j}$ is the p'^{th} action that writes to s, by Lemma 4, we can get that the corresponding action of $\alpha'_{i:j}$ in δ is also the p'^{th} action that writes to s. As Condition I holds, we know $\alpha_{i:j}$ is the corresponding action of $\alpha_{i:j}$ in δ'. Since $\alpha_{i:j}$ operates on s with version number p in our assumption, we get $p' = p$. Otherwise if $\alpha'_{i:j}$ reads on s, by Lemma 5, we can get that $\alpha'_{i:j}$'s corresponding action in δ also reads s with the same version number, and similarly, we get $p' = p$.

We next prove both Condition I and Condition II hold. For condition I, suppose $\alpha_{i:j-1}$ and $\alpha'_{i:j-1}$ operate on the variable $s1$ with version number $p1$. To satisfy condition I, we need again to make sure (Ia) $\pi'_{i:j-2} = \pi_{i:j-2}$ and (Ib) $\sigma'[s1]^{p1} = \sigma[s1]^{p1}$. For condition II, let $\alpha_{i1:j1}$ and $\alpha'_{i1':j1'}$ denote the actions that write $\sigma[s]^p$ and $\sigma'[s]^p$, respectively. Since the current value of a variable is determined by the action that last writes to it, to satisfy condition II, we need to make sure $\alpha'_{i1':j1'}$ is equal to $\alpha_{i1:j1}$, which again requires (IIa) $\pi'_{i1':j1'-1} = \pi_{i1:j1-1}$ and (IIb) $\sigma'[s]^{p-1} = \sigma[s]^{p-1}$. If we apply this reasoning logic deductively for all threads, we will finally reach the base condition (i) $\forall t_i \in \mathbb{T}, \pi'_{i:0} = \pi_{i:0}$ and (ii) $\forall s \in \mathbb{S}, \sigma'[s]^0 = \sigma[s]^0$, which are satisfied by the equivalence of the initial program states $\Sigma'^0 = \Sigma^0$. Hence, Lemma 3 is proved.

Therefore, Theorem 1 is proved. ■

A Family of Abstract Interpretations for Static Analysis of Concurrent Higher-Order Programs

Matthew Might and David Van Horn

University of Utah and Northeastern University
might@cs.utah.edu, dvanhorn@ccs.neu.edu
http://matt.might.net/, http://lambda-calcul.us/

Abstract. We develop a framework for computing two foundational analyses for *concurrent* higher-order programs: (control-)flow analysis (CFA) and may-happen-in-parallel analysis (MHP). We pay special attention to the unique challenges posed by the unrestricted mixture of first-class continuations and dynamically spawned threads. To set the stage, we formulate a concrete model of concurrent higher-order programs: the P(CEK*)S machine. We find that the systematic abstract interpretation of this machine is capable of computing both flow and MHP analyses. Yet, a closer examination finds that the precision for MHP is poor. As a remedy, we adapt a shape analytic technique—singleton abstraction—to dynamically spawned threads (as opposed to objects in the heap). We then show that if MHP analysis is not of interest, we can substantially accelerate the computation of flow analysis alone by collapsing thread interleavings with a second layer of abstraction.

1 Higher-Order Is Hard; Concurrency Makes It Harder

The next frontier in static reasoning for higher-order programs is concurrency. When unrestricted concurrency and higher-order computation meet, their challenges to static reasoning reinforce and amplify one another.

Consider the possibilities opened by a mixture of dynamically created threads and first-class continuations. Both pose obstacles to static analysis by themselves, yet the challenge of reasoning about a continuation created in one thread and invoked in another is substantially more difficult than the sum of the individual challenges.

We respond to the challenge by (1) constructing the P(CEK*)S machine, a nondeterministic abstract machine that concretely and fully models higher-orderness and concurrency; and then (2) systematically deriving abstract interpretations of this machine to enable the sound and meaningful flow analysis of concurrent higher-order programs.

Our first abstract interpretation creates a dual hierarchy of flow and may-happen-in-parallel (MHP) analyses parameterized by context-sensitivity and the granularity of an abstract partition among threads. The context-sensitivity knob tunes flow-precision as in Shivers's k-CFA [21]. The partition among threads tunes the precision of MHP analysis, since it controls the mapping of concrete

E. Yahav (Ed.): SAS 2011, LNCS 6887, pp. 180–197, 2011.

threads onto abstract threads. To improve the precision of MHP analysis, our second abstract interpretation introduces shape analytic concepts—chiefly, singleton cardinality analysis—but it applies them to discover the "shape" of threads rather than the shape of objects in the heap. The final abstract interpretation accelerates the computation of flow analysis (at the cost of MHP analysis) by inflicting a second abstraction that soundly collapses all thread interleavings together.

1.1 Challenges to Reasoning about Higher-Order Concurrency

The combination of higher-order computation and concurrency introduces design patterns that challenge conventional static reasoning techniques.

Challenge: Optimizing Futures. Futures are a popular means of enabling parallelism in functional programming. Expressions marked `future` are computed in parallel with their own continuation. When that value reaches a point of strict evaluation, the thread of the continuation joins with the thread of the future.

Unfortunately, the standard implementation of futures [5] inflicts substantial costs on sequential performance: that implementation transforms `(future` e`)` into `(spawn` e`)`, and all strict expressions into conditionals and thread-joins. That is, if the expression e' is in a strict evaluation position, then it becomes:

```
(let ([$t e']) (if (thread? $t) (join $t) $t))
```

Incurring this check at all strict points is costly. A flow analysis that works for concurrent programs would find that most expressions can never evaluate to future value, and thus, need not incur such tests.

Challenge: Thread Cloning/Replication. The higher-order primitive `call/cc` captures the current continuation and passes it as a first-class value to its argument. The primitive `call/cc` is extremely powerful—a brief interaction between `spawn` and `call/cc` effortlessly expresses thread replication:

```
(call/cc (lambda (cc) (spawn (cc #t)) #f))
```

This code captures the current continuation, spawns a new thread and replicates the spawning thread in the spawned thread by invoking that continuation. The two threads can be distinguished by the return value of `call/cc`: the replicant returns true and the original returns false.

Challenge: Thread Metamorphosis. Consider a web server in which continuations are used to suspend and restore computations during interactions with the client [18]. Threads "morph" from one kind of thread (an interaction thread or a worker thread) to another by invoking continuations. The `begin-worker` continuation metamorphizes the calling thread into a worker thread:

```
(define become-worker
 (let ([cc (call/cc (lambda (cc) (cc cc)))])
  (cond
   [(continuation? cc)       cc]
   [else                     (handle-next-request)
                             (become-worker #t)])))
```

The procedure handle-next-request checks whether the request is the resumption of an old session, and if so, invokes the continuation of that old session:

```
(define (handle-next-request)
 (define request (next-request))
 (atomic-hash-remove! (session-id request)
  (lambda (session-continuation)
    (define answer (request->answer request))
    (session-continuation answer))
  (lambda () (start-new-session request))))
```

When a client-handling thread needs data from the client, it calls read-from-client, it associates the current continuation to the active session, piggy-backs a request to the client on an outstanding reply and the metamorphizes into a worker thread to handle other incoming clients:

```
(define (read-from-client session)
 (call/cc (lambda (cc)
  (atomic-hash-set! sessions (session-id session) cc)
  (reply-to session))
  (become-worker #t)))
```

2 P(CEK*)S: An Abstract Machine Model of Concurrent, Higher-Order Computation

In this section, we define a P(CEK*)S machine—a CESK machine with a pointer refinement that allows concurrent threads of execution. It is directly inspired by the sequential abstract machines in Van Horn and Might's recent work [23]. Abstract interpretations of this machine perform both flow and MHP analysis for concurrent, higher-order programs.

The language modeled in this machine (Figure 1) is A-Normal Form lambda calculus [9] augmented with a core set of primitives for multithreaded programming. For concurrency, it features an atomic compare-and-swap operation, a spawn form to create a thread from an expression and a join operation to wait for another thread to complete. For higher-order computation, it features closures and first-class continuations. A closure is a first-class procedure constructed by pairing a lambda term with an environment that fixes the meaning of its free variables. A continuation reifies the sequential control-flow for the remainder of the thread as a value; when a continuation is "invoked," it restores that control-flow. Continuations may be invoked an arbitrary number of times, and at any time since their moment of creation.

$$\begin{aligned}
e \in \mathsf{Exp} &::= \texttt{(let ((}v\ \mathit{cexp}\texttt{))}\ e\texttt{)} \\
&\mid\ \mathit{cexp} \\
&\mid\ \textit{\ae} \\
\mathit{cexp} \in \mathsf{CExp} &::= \texttt{(}f\ \textit{\ae}_1 \ldots \textit{\ae}_n\texttt{)} \\
&\mid\ \texttt{(callcc}\ \textit{\ae}\texttt{)} \\
&\mid\ \texttt{(set!}\ v\ \textit{\ae}_{\mathrm{value}}\texttt{)} \\
&\mid\ \texttt{(if}\ \textit{\ae}\ \mathit{cexp}\ \mathit{cexp}\texttt{)} \\
&\mid\ \texttt{(cas}\ v\ \textit{\ae}_{\mathrm{old}}\ \textit{\ae}_{\mathrm{new}}\texttt{)} \\
&\mid\ \texttt{(spawn}\ e\texttt{)} \\
&\mid\ \texttt{(join}\ \textit{\ae}\texttt{)} \\
f, \textit{\ae} \in \mathsf{AExp} &::= \mathit{lam} \mid v \mid n \mid \texttt{\#f} \\
\mathit{lam} \in \mathsf{Lam} &::= \texttt{(}\lambda\ \texttt{(}v_1 \ldots v_n\texttt{)}\ e\texttt{)}
\end{aligned}$$

Fig. 1. ANF lambda-calculus augmented with a core set of primitives for concurrency

2.1 P(CEK*)S: A Concrete State-Space

A concrete state of execution in the P(CEK*)S machine contains a set of threads plus a shared store. Each thread is a context combined with a thread id. A context contains the current expression, the current environment, an address pointing to the current continuation, and a thread history:

$$\begin{aligned}
\varsigma \in \Sigma &= \mathit{Threads} \times \mathit{Store} \\
T \in \mathit{Threads} &= \mathcal{P}\left(\mathit{Context} \times \mathit{TID}\right) \\
c \in \mathit{Context} &= \mathsf{Exp} \times \mathit{Env} \times \mathit{Addr} \times \mathit{Hist} \\
\rho \in \mathit{Env} &= \mathsf{Var} \rightharpoonup \mathit{Addr} \\
\kappa \in \mathit{Kont} &= \mathsf{Var} \times \mathsf{Exp} \times \mathit{Env} \times \mathit{Addr} + \{\mathbf{halt}\} \\
h \in \mathit{Hist} &\text{ contains records of thread history} \\
\sigma \in \mathit{Store} &= \mathit{Addr} \to D \\
d \in D &= \mathit{Value} \\
\mathit{val} \in \mathit{Value} &= \mathit{Clo} + \mathit{Bool} + \mathit{Num} + \mathit{Kont} + \mathit{TID} + \mathit{Addr} \\
\mathit{clo} \in \mathit{Clo} &= \mathsf{Lam} \times \mathit{Env} \\
a \in \mathit{Addr} &\text{ is an infinite set of addresses} \\
t \in \mathit{TID} &\text{ is an infinite set of thread ids.}
\end{aligned}$$

The P(CEK*)S machine allocates continuations in the store; thus, to add first-class continuations, we have first-class addresses. Under abstraction, program history determines the context-sensitivity of an individual thread. To allow context-sensitivity to be set as external parameter, we'll leave program history opaque. (For example, to set up a k-CFA-like analysis, the program history would be the sequence of calls made since the start of the program.) To parameterize the precision of MHP analysis, the thread ids are also opaque.

2.2 P(CEK*)S: A Factored Transition Relation

Our goal is to factor the semantics of the P(CEK*)S machine, so that one can drop in a classical CESK machine to model sequential language features. The abstract interpretation maintains the same factoring, so that existing analyses of higher-order programs may be "plugged into" the framework for handling concurrency. The relation ($\Rightarrow$) models concurrent transition, and the relation ($\rightarrow$) models sequential transition:

$$\begin{aligned}(\Rightarrow) &\subseteq \Sigma \times \Sigma \\ (\rightarrow) &\subseteq (\mathit{Context} \times \mathit{Store}) \times (\mathit{Context} \times \mathit{Store})\end{aligned}$$

For instance, the concurrent transition relation invokes the sequential transition relation to handle `if`, `set!`, `cas`, `callcc` or procedure call:[1]

$$\frac{(c, \sigma) \rightarrow (c', \sigma')}{(\{(c,t)\} \uplus T, \sigma) \Rightarrow (\{(c',t)\} \cup T, \sigma')}$$

Given a program e, the injection function $\mathcal{I} : \mathsf{Exp} \rightarrow \mathit{State}$ creates the initial machine state:

$$\mathcal{I}(e) = (\{((e, [], a_{\mathbf{halt}}, h_0), t_0)\}, [a_{\mathbf{halt}} \mapsto \mathbf{halt}]),$$

where t_0 is the distinguished initial thread id, h_0 is a blank history and $a_{\mathbf{halt}}$ is the distinguished address of the **halt** continuation. The meaning of a program e is the (possibly infinite) set of states reachable from the initial state:

$$\{\varsigma : \mathcal{I}(e) \Rightarrow^* \varsigma\}.$$

Sequential Transition Example: ***callcc*** There are ample resources dating to Felleisen and Friedman [6] detailing the transition relation of a CESK machine. For a recent treatment that covers both concrete and abstract transition, see Van Horn and Might [23]. Most of the transitions are straightforward, but in the interest of more self-containment, we review the callcc transition:

$$\begin{aligned}(\overbrace{(\llbracket\texttt{(callcc } \text{æ}\texttt{)}\rrbracket, \rho, a_\kappa, h)}^{c}, \sigma) &\Rightarrow ((e, \rho'', a_\kappa, h'), \sigma'), \text{ where} \\ h' &= \mathit{record}(c, h) \\ (\llbracket\texttt{(}\lambda\ \texttt{(}v\texttt{)}\ e\texttt{)}\rrbracket, \rho') &= \mathcal{E}(\text{æ}, \rho, \sigma) \\ a &= \mathit{alloc}(v, h') \\ \rho'' &= \rho'[v \mapsto a] \\ \sigma' &= \sigma[a \mapsto a_\kappa].\end{aligned}$$

[1] The transition for `cas` is "sequential" in the sense that its action is atomic.

The atomic evaluation function $\mathcal{E} : \mathsf{AExp} \times \mathit{Env} \times \mathit{Store} \rightharpoonup D$ maps an atomic expression to a value in the context of an environment and a store; for example:

$$\mathcal{E}(v, \rho, \sigma) = \sigma(\rho(v))$$
$$\mathcal{E}(\mathit{lam}, \rho, \sigma) = (\mathit{lam}, \rho).$$

(The notation $f[x \mapsto y]$ is functional extension: the function identical to f, except that x now yields y instead of $f(x)$.)

2.3 A Shift in Perspective

Before proceeding, it is worth shifting the formulation so as to ease the process of abstraction. For instance, the state-space is well-equipped to handle a finite abstraction over addresses, since we can promote the range of the store to *sets* of values. This allows multiple values to live at the same address once an address has been re-allocated. The state-space is less well-equipped to handle the approximation on thread ids. When abstracting thread ids, we could keep a set of abstract threads paired with the store. But, it is natural to define the forthcoming concrete and abstract transitions when the set of threads becomes a map. Since every thread has a distinct thread id, we can model the set of threads in each state as a partial map from a thread id to a context:

$$\mathit{Threads} \equiv \mathit{TID} \rightharpoonup \mathit{Context}.$$

It is straightforward to update the concurrent transition relation when it calls out to the sequential transition relation:

$$\frac{(c, \sigma) \to (c', \sigma')}{(T[t \mapsto c], \sigma) \Rightarrow (T[t \mapsto c'], \sigma').}$$

2.4 Concurrent Transition in the P(CEK⋆)S Machine

We define the concurrent transitions separately from the sequential transitions. For instance, if a context is attempting to `spawn` a thread, the concurrent relation handles it by allocating a new thread id t', and binding it to the new context c'':

$$\begin{aligned}
(T[t \mapsto \overbrace{([\![(\texttt{spawn}\ e)]\!], \rho, a_\kappa, h)}^{c}], \sigma) &\Rightarrow (T[t \mapsto c', t' \mapsto c''], \sigma'),\\
\text{where } t' &= \mathit{newtid}(c, T[t \mapsto c])\\
c'' &= (e, \rho, a_{\mathbf{halt}}, h_0)\\
h' &= \mathit{record}(c, h)\\
(v', e', \rho', a'_\kappa) &= \sigma(a_\kappa)\\
a' &= \mathit{alloc}(v', h')\\
\rho'' &= \rho'[v' \mapsto a']\\
c' &= (e', \rho'', a'_\kappa, h')\\
\sigma' &= \sigma[a' \mapsto t'], \text{ where:}
\end{aligned}$$

- *newtid* : *Context* × *Threads* → *TID* allocates a fresh thread id for the newly spawned thread.
- *record* : *Context* × *Hist* → *Hist* is responsible for updating the history of execution with this context.
- *alloc* : Var × *Hist* → *Addr* allocates a fresh address for the supplied variable.

The abstract counterparts to these functions determine the degree of approximation in the analysis, and consequently, the trade-off between speed and precision.

When a thread halts, its thread id is treated as an address, and its return value is stored there:

$$(T[t \mapsto ([\![æ]\!], \rho, a_{\mathbf{halt}}, h)], \sigma) \Rightarrow (T, \sigma[t \mapsto \mathcal{E}(æ, \rho, \sigma)]).$$

This convention, of using thread ids as addresses, makes it easy to model thread joins, since they can check to see if that address has value waiting or not:

$$\frac{\sigma(\mathcal{E}(æ, \rho, \sigma)) = d}{(T[t \mapsto \underbrace{([\![\texttt{(join}\ æ\texttt{)}]\!], \rho, a_\kappa, h)}_{c}], \sigma) \Rightarrow (T[t \mapsto (e, \rho', a'_\kappa, h')], \sigma'),}$$

$$\begin{aligned} \text{where } \kappa &= \sigma(a_\kappa) \\ (v, e, \rho, a'_\kappa) &= \kappa \\ \rho' &= \rho[v \mapsto a''] \\ h' &= \mathit{record}(c, h) \\ a'' &= \mathit{alloc}(v, h') \\ \sigma' &= \sigma[a'' \mapsto d]. \end{aligned}$$

3 A Systematic Abstract Interpretation of P(CEK*)S

Using the techniques outlined in our recent work on systematically constructing abstract interpretations from abstract machines [23], we can directly convert the P(CEK*)S machine into an abstract interpretation of itself. In the concrete state-space, there are four points at which we must inflict abstraction: over basic values (like numbers), over histories, over addresses and over thread ids.

The abstraction over histories determines the context-sensitivity of the analysis on a per-thread basis. The abstraction over addresses determines polyvariance. The abstraction over thread ids maps concrete threads into abstract threads, which determines to what extent the analysis can distinguish dynamically created threads from one another; it directly impacts MHP analysis.

The abstract state-space (Figure 2) mirrors the concrete state-space in structure. We assume the natural point-wise, element-wise and member-wise lifting of a partial order ($\sqsubseteq$) over all of the sets within the state-space. Besides the restriction of histories, addresses and thread ids to finite sets, it is also worth

$$\begin{aligned}
\hat{\varsigma} \in \hat{\Sigma} &= \widehat{\mathit{Threads}} \times \widehat{\mathit{Store}} \\
\hat{T} \in \widehat{\mathit{Threads}} &= \widehat{\mathit{TID}} \to \mathcal{P}(\widehat{\mathit{Context}}) \\
\hat{c} \in \widehat{\mathit{Context}} &= \mathsf{Exp} \times \widehat{\mathit{Env}} \times \widehat{\mathit{Addr}} \times \widehat{\mathit{Hist}} \\
\hat{\rho} \in \widehat{\mathit{Env}} &= \mathsf{Var} \rightharpoonup \widehat{\mathit{Addr}} \\
\hat{\kappa} \in \widehat{\mathit{Kont}} &= \mathsf{Var} \times \mathsf{Exp} \times \widehat{\mathit{Env}} \times \widehat{\mathit{Addr}} + \{\mathbf{halt}\} \\
\hat{h} \in \widehat{\mathit{Hist}} &\text{ contains bounded, finite program histories} \\
\hat{\sigma} \in \widehat{\mathit{Store}} &= \widehat{\mathit{Addr}} \to \hat{D} \\
\hat{d} \in \hat{D} &= \mathcal{P}(\widehat{\mathit{Value}}) \\
\widehat{\mathit{val}} \in \widehat{\mathit{Value}} &= \widehat{\mathit{Clo}} + \mathit{Bool} + \widehat{\mathit{Num}} + \widehat{\mathit{Kont}} + \widehat{\mathit{TID}} + \widehat{\mathit{Addr}} \\
\widehat{\mathit{clo}} \in \widehat{\mathit{Clo}} &= \mathsf{Lam} \times \widehat{\mathit{Env}} \\
\hat{a} \in \widehat{\mathit{Addr}} &\text{ is a finite set of abstract addresses} \\
\hat{t} \in \widehat{\mathit{TID}} &\text{ is a finite set of abstract thread ids}
\end{aligned}$$

Fig. 2. Abstract state-space for a systematic abstraction of the P(CEK*)S machine

pointing out that the range of both $\widehat{\mathit{Threads}}$ and $\widehat{\mathit{Store}}$ are *power* sets. This promotion occurs because, during the course of an analysis, re-allocating the same thread id or address is all but inevitable. To maintain soundness, the analysis must be able to store multiple thread contexts in the same abstract thread id, and multiple values at the same address in the store.

The structural abstraction map α on the state-space (Figure 3) utilizes a family of abstraction maps over the sets within the state-space. With the abstraction and the abstract state-space fixed, the abstract transition relation reduces to a matter of calculation [4]. The relation $(\leadsto)$ describes the concurrent abstract transition, while the relation $(\multimap)$ describes the sequential abstract transition:

$$\begin{aligned}
(\leadsto) &\subseteq \hat{\Sigma} \times \hat{\Sigma} \\
(\multimap) &\subseteq (\widehat{\mathit{Context}} \times \widehat{\mathit{Store}}) \times (\widehat{\mathit{Context}} \times \widehat{\mathit{Store}})
\end{aligned}$$

When the context in focus is sequential, the sequential relation takes over:

$$\frac{(\hat{c}, \hat{\sigma}) \multimap (\hat{c}', \hat{\sigma}')}{(\hat{T}[\hat{t} \mapsto \{\hat{c}\} \cup \hat{C}], \hat{\sigma}) \leadsto (\hat{T} \sqcup [\hat{t} \mapsto \{\hat{c}'\}], \hat{\sigma}')}$$

There is a critical change over the concrete rule in this abstract rule: thanks to the join operation, the abstract context remains associated with the abstract thread id even after its transition has been considered. In the next section, we will examine the application of singleton abstraction to thread ids to allow the "strong update" of abstract threads ids across transition. (For programs whose

$$\begin{aligned}
\alpha_{\Sigma}(T,\sigma) &= (\alpha(T), \alpha(\sigma))\\
\alpha_{\mathit{Threads}}(T) &= \lambda\hat{t}.\bigsqcup_{\alpha(t)=\hat{t}} \alpha(T(t))\\
\alpha_{\mathit{Context}}(e,\rho,\kappa,h) &= \{(e,\alpha(\rho),\alpha(\kappa),\alpha(h))\}\\
\alpha_{\mathit{Env}}(\rho) &= \lambda v.\alpha(\rho(v))\\
\alpha_{\mathit{Kont}}(v,e,\rho,a) &= (v,e,\alpha(\rho),\alpha(a))\\
\alpha_{\mathit{Kont}}(\mathbf{halt}) &= \mathbf{halt}\\
\alpha_{\mathit{Store}}(\sigma) &= \lambda\hat{a}.\bigsqcup_{\alpha(a)=\hat{a}} \alpha(\sigma(a))\\
\alpha_{D}(\mathit{val}) &= \{\alpha(\mathit{val})\}\\
\alpha_{\mathit{Clo}}(\mathit{lam},\rho) &= (\mathit{lam},\alpha(\rho))\\
\alpha_{\mathit{Bool}}(b) &= b\\
\alpha_{\mathit{Hist}}(h) &\text{ is defined by context-sensitivity}\\
\alpha_{\mathit{TID}}(t) &\text{ is defined by thread-sensitivity}\\
\alpha_{\mathit{Addr}}(a) &\text{ is defined by polyvariance.}
\end{aligned}$$

Fig. 3. A structural abstraction map

maximum number of threads is statically bounded by a known constant, this allows for precise MHP analysis.)

3.1 Running the Analysis

Given a program e, the injection function $\hat{\mathcal{I}} : \mathsf{Exp} \to \widehat{State}$ creates the initial abstract machine state:

$$\hat{\mathcal{I}}(e) = \left(\left[\hat{t}_0 \mapsto \left\{(e, [], \hat{a}_{\mathbf{halt}}, \hat{h}_0)\right\}\right], [\hat{a}_{\mathbf{halt}} \mapsto \{\mathbf{halt}\}]\right),$$

where $\hat{h}_0$ is a blank abstract history and $\hat{a}_{\mathbf{halt}}$ is the distinguished abstract address of the **halt** continuation. The analysis of a program e is the finite set of states reachable from the initial state:

$$\hat{\mathcal{R}}(e) = \left\{\hat{\varsigma} : \hat{\mathcal{I}}(e) \leadsto^* \hat{\varsigma}\right\}.$$

3.2 A Fixed Point Interpretation

If one prefers a traditional, fixed-point abstract interpretation, we can imagine the intermediate state of the analysis itself as a set of currently reachable abstract machine states:

$$\hat{\xi} \in \hat{\Xi} = \mathcal{P}(\hat{\Sigma}).$$

A global transfer function $\hat{f} : \hat{\Xi} \to \hat{\Xi}$ evolves this set:

$$\hat{f}(\hat{\xi}) = \left\{\hat{\mathcal{I}}(e)\right\} \cup \left\{\hat{\varsigma}' : \hat{\varsigma} \in \hat{\xi} \text{ and } \hat{\varsigma} \leadsto \hat{\varsigma}'\right\}.$$

The solution of the analysis is the least fixed point: $\text{lfp}(\hat{f})$.

3.3 Termination

The dependence structure of the abstract state-space is a directed acyclic graph starting from the set $\hat{\Sigma}$ at the root. Because all of the leaves of this graph (*e.g.*, lambda terms, abstract numbers, abstract addresses) are finite for any given program, the state-space itself must also be finite. Consequently, there are no infinitely ascending chains in the lattice $\hat{\Xi}$. By Kleene's fixed point theorem, there must exist a least natural n such that $\text{lfp}(\hat{f}) = \hat{f}^n(\emptyset)$.

3.4 Concurrent Abstract Transitions

Guided by the structural abstraction, we can convert the concrete concurrent transitions for the P(CEK*)S machine into concurrent abstract transitions. For instance, if an abstract context is attempting to `spawn` a thread, the concurrent relation handles it by allocating a new thread id $\hat{t}'$, and binding it to the new context $\hat{c}''$:

$$(\hat{T}[\hat{t} \mapsto \{\overbrace{(\llbracket\texttt{(spawn}\ e\texttt{)}\rrbracket, \hat{\rho}, \hat{a}_{\hat{\kappa}}, \hat{h})}^{\hat{c}}\} \cup \hat{C}], \hat{\sigma}) \leadsto (\hat{T} \sqcup [\hat{t} \mapsto \{\hat{c}'\}, \hat{t}' \mapsto \{\hat{c}''\}], \sigma'),$$

$$\begin{aligned}
\text{where } \hat{t}' &= \widehat{newtid}(\hat{c}, \hat{T}[\hat{t} \mapsto \hat{C} \cup \{\hat{c}\}]) \\
\hat{c}'' &= (e, \hat{\rho}, \hat{a}_{\mathbf{halt}}, \hat{h}_0) \\
\hat{h}' &= \widehat{record}(\hat{c}, \hat{h}) \\
(v', e', \hat{\rho}', \hat{a}'_{\hat{\kappa}}) &\in \hat{\sigma}(\hat{a}_{\hat{\kappa}}) \\
\hat{a}' &= \widehat{alloc}(v', \hat{h}') \\
\hat{\rho}'' &= \hat{\rho}'[v' \mapsto \hat{a}'] \\
\hat{c}' &= (e', \hat{\rho}'', \hat{a}'_{\hat{\kappa}}, \hat{h}') \\
\hat{\sigma}' &= \hat{\sigma} \sqcup [\hat{a}' \mapsto \{\hat{t}'\}], \text{ where:}
\end{aligned}$$

- $\widehat{newtid} : \widehat{Context} \times \widehat{Threads} \to \widehat{TID}$ allocates a thread id for the newly spawned thread.
- $\widehat{record} : \widehat{Context} \times \widehat{Hist} \to \widehat{Hist}$ is responsible for updating the (bounded) history of execution with this context.
- $\widehat{alloc} : \mathsf{Var} \times \widehat{Hist} \to \widehat{Addr}$ allocates an address for the supplied variable.

These functions determine the degree of approximation in the analysis, and consequently, the trade-off between speed and precision.

When a thread halts, its abstract thread id is treated as an address, and its return value is stored there:

$$\frac{\hat{T}' = \hat{T} \sqcup [\hat{t} \mapsto \{(\llbracket æ \rrbracket, \hat{\rho}, \hat{a}_{\mathbf{halt}}, \hat{h})\}]}{(\hat{T}', \sigma) \leadsto (\hat{T}', \hat{\sigma} \sqcup [\hat{t} \mapsto \hat{\mathcal{E}}(æ, \hat{\rho}, \hat{\sigma})])}, \text{ where}$$

the atomic evaluation function $\hat{\mathcal{E}} : \mathsf{AExp} \times \widehat{Env} \times \widehat{Store} \rightarrow \hat{D}$ maps an atomic expression to a value in the context of an environment and a store:

$$\hat{\mathcal{E}}(v, \hat{\rho}, \hat{\sigma}) = \hat{\sigma}(\hat{\rho}(v))$$
$$\hat{\mathcal{E}}(lam, \hat{\rho}, \hat{\sigma}) = \{(lam, \hat{\rho})\}.$$

It is worth asking whether it is sound in just this case to remove the context from the threads (making the subsequent threads T instead of T'). It is sound, but it seems to require a (slightly) more complicated staggered-state bisimulation to prove it: the concrete counterpart to this state may take several steps to eliminate all of its halting contexts.

Thanks to the convention to use thread ids as addresses holding the return value of thread, it easy to model thread joins, since they can check to see if that address has a value waiting or not:

$$\frac{\hat{a} \in \hat{\mathcal{E}}(æ, \hat{\rho}, \hat{\sigma}) \qquad \hat{d} = \hat{\sigma}(\hat{a})}{(\hat{T} \sqcup [\hat{t} \mapsto \{\underbrace{(\llbracket \texttt{(join } æ\texttt{)} \rrbracket, \hat{\rho}, \hat{a}_{\hat{\kappa}}, \hat{h})}_{\hat{c}}\}], \hat{\sigma}) \leadsto (\hat{T} \sqcup [\hat{t} \mapsto \{(e, \hat{\rho}', \hat{a}'_{\hat{\kappa}}, \hat{h}'), \hat{c}\}], \hat{\sigma}'),}$$

$$\begin{aligned} \text{where } \hat{\kappa} &\in \hat{\sigma}(\hat{a}_{\hat{\kappa}}) \\ (v, e, \hat{\rho}, \hat{a}'_{\hat{\kappa}}) &= \hat{\kappa} \\ \hat{\rho}' &= \hat{\rho}[v \mapsto \hat{a}''] \\ \hat{h}' &= \widehat{record}(\hat{c}, \hat{h}) \\ \hat{a}'' &= \widehat{alloc}(v, \hat{h}') \\ \hat{\sigma}' &= \hat{\sigma} \sqcup [\hat{a}'' \mapsto \hat{d}]. \end{aligned}$$

3.5 Soundness

Compared to a standard proof of soundness for a small-step abstract interpretation, the proof of soundness requires only slightly more attention in a concurrent setting. The key lemma in the inductive proof of simulation states that when a concrete state ς abstracts to an abstract state $\hat{\varsigma}$, if the concrete state can transition to ς', then the abstract state $\hat{\varsigma}$ must be able to transition to some other abstract state $\hat{\varsigma}'$ such that ς' abstracts to $\hat{\varsigma}'$:

Theorem 1. *If:*

$$\alpha(\varsigma) \sqsubseteq \hat{\varsigma} \text{ and } \varsigma \Rightarrow \varsigma',$$

then there must exist a state $\hat{\varsigma}'$ *such that:*

$$\alpha(\varsigma') \sqsubseteq \hat{\varsigma}' \text{ and } \hat{\varsigma} \leadsto \hat{\varsigma}'.$$

Proof. The proof is follows the case-wise structure of proofs like those in Might and Shivers [14]. There is an additional preliminary step: first choose a thread id modified across transition, and then perform case-wise analysis on how it could have been modified.

3.6 Extracting Flow Information

The core question in flow analysis is, "Can the value *val* flow to the expression *æ*?" To answer it, assume that $\hat{\xi}$ is the set of all reachable abstract states. We must check every state within this set, and every environment $\hat{\rho}$ within that state. If $\hat{\sigma}$ is the store in that state, then *val* may flow to *æ* if the value $\alpha(val)$ is represented in the set $\hat{\mathcal{E}}(æ, \hat{\rho}, \hat{\sigma})$. Formally, we can construct a flows-to relation, $\mathit{FlowsTo} \subseteq \mathit{Value} \times \mathsf{AExpr}$, for a program e:

$$\mathit{FlowsTo}(val, æ) \text{ iff there exist } (\hat{T}, \hat{\sigma}) \in \hat{\mathcal{R}}(e) \text{ and } \hat{t} \in \mathit{dom}(\hat{T}) \text{ such that}$$
$$(e, \hat{\rho}, \hat{\kappa}) \in \hat{T}(\hat{t}) \text{ and } \{\alpha(val)\} \sqsubseteq \hat{\mathcal{E}}(æ, \hat{\rho}, \hat{\sigma}).$$

3.7 Extracting MHP Information

In MHP analysis, we are concerned with whether two expressions e' and e'' may be evaluated concurrently with one another in program e. It is straightforward to decide this using the set of reachable states computed by the abstract interpretation. If, in any reachable state, there exist two distinct contexts at the relevant expressions, then their evaluation may happen in parallel with one another. Formally, the $\mathit{MHP} \subseteq \mathsf{Exp} \times \mathsf{Exp}$ relation with respect to program e is:

$$\mathit{MHP}(e', e'') \text{ iff there exist } (\hat{T}, \hat{\sigma}) \in \hat{\mathcal{R}}(e) \text{ and } \hat{t}', \hat{t}'' \in \mathit{dom}(\hat{T}) \text{ such that}$$
$$(e', \_, \_, \_) \in \hat{T}(\hat{t}') \text{ and } (e'', \_, \_, \_) \in \hat{T}(\hat{t}'').$$

4 MHP: Making Strong Transitions with Singleton Threads

In the previous section, we constructed a systematic abstraction of the P(CEK*)S machine. While it serves as a sound and capable flow analysis, its precision as an MHP analysis is just above useless. Contexts associated with each abstract thread id grow monotonically during the course of the analysis. Eventually, it will seem as though every context may happen in parallel with every other context.

By comparing the concrete and abstract semantics, the cause of the imprecision becomes clear: where the concrete semantics *replaces* the context at a given thread id, the abstract semantics *joins*.

Unfortunately, we cannot simply discard the join. A given abstract thread id could be representing multiple concrete thread ids. Discarding a thread id would then discard possible interleavings, and it could even introduce unsoundness.

Yet, it is plainly the case that *many* programs have a boundable number of threads that are co-live. Thread creation is considered expensive, and thread pools created during program initialization are a popular mechanism for circumventing the problem. To exploit this design pattern, we can make thread ids eligible for "strong update" across transition. In shape analysis, strong update refers to the ability to treat an abstract address as the representative of a single concrete address when assigning to that address. That is, by tracking the cardinality of the abstraction of each thread id, we can determine when it is sound to replace functional join with functional update *on threads themselves*.

The necessary machinery is straightforward, adapted directly from the shape analysis literature [1,2,11,10,14,19] ; we attach to each state a cardinality counter $\hat{\mu}$ that tracks how many times an abstract thread id has been allocated (but not precisely beyond once):

$$\hat{\varsigma} \in \hat{\Sigma} = \widehat{Threads} \times \widehat{Store} \times \widehat{TCount}$$
$$\hat{\mu} \in \widehat{TCount} = \widehat{TID} \rightarrow \{0, 1, \infty\}\,.$$

When the count of an abstract thread id is exactly one, we know for certain that there exists at most one concrete counterpart. Consequently, it is safe to perform a "strong transition." Consider the case where the context in focus for the concurrent transition is sequential; in the case where the count is exactly one, the abstract context gets replaced on transition:

$$\frac{(\hat{c}, \hat{\sigma}) \multimap (\hat{c}', \hat{\sigma}') \qquad \hat{\mu}(\hat{t}) = 1}{(\hat{T}[\hat{t} \mapsto \{\hat{c}\} \uplus \hat{C}], \hat{\sigma}, \hat{\mu}) \leadsto (\hat{T}[\hat{t} \mapsto \{\hat{c}'\} \cup \hat{C}], \hat{\sigma}', \hat{\mu}).}$$

It is straightforward to modify the existing concurrent transition rules to exploit information available in the cardinality counter. At the beginning of the analysis, all abstract thread ids have a count of zero. Upon spawning a thread, the analysis increments the result of the function $\widehat{newtid}$. When a thread whose abstract thread id has a count of one halts, its count is reset to zero.

4.1 Strategies for Abstract Thread id Allocation

Just as the introduction of an allocation function for addresses provides the ability to tune polyvariance, the $\widehat{newtid}$ function provides the ability to tune precision. The optimal strategy for allocating this scarce pool of abstract thread ids depends upon the design patterns in use.

One could, for instance, allocate abstract thread ids according to calling context, *e.g.*, the abstract thread id is the last k call sites. This strategy would work well for the implementation of futures, where futures from the same context are often not co-live with themselves.

The context-based strategy, however, is not a reasonable strategy for a thread-pool design pattern. All of the spawns will occur at the same expression in the same loop, and therefore, in the same context. Consequently, there will be no discrimination between threads. If the number of threads in the pool is known *a priori* to be n, then the right strategy for this pattern is to create n abstract thread ids, and to allocate a new one for each iteration of the thread-pool-spawning loop. On the other hand, if the number of threads is set dynamically, no amount of abstract thread ids will be able to discriminate between possible interleavings effectively, in this case a reasonable choice for precision would be to have one abstract thread per thread pool.

4.2 Advantages for MHP Analysis

With the cardinality counter, it is possible to test whether an expression may be evaluated in parallel with itself. If, for every state, every abstract thread id which maps to a context containing that expression has a count of one, and no other context contains that expression, then that expression must never be evaluated in parallel with itself. Otherwise, parallel evaluation is possible.

5 Flow Analysis of Concurrent Higher-Order Programs

If the concern is a sound flow analysis, but not MHP analysis, then we can perform an abstract interpretation of our abstract interpretation that efficiently collapses all possible interleavings and paths, even as it retains limited (reachability-based) flow-sensitivity. This second abstraction map $\alpha' : \hat{\Xi} \to \hat{\Sigma}$ operates on the system-space of the fixed-point interpretation:

$$\alpha'(\hat{\xi}) = \bigsqcup_{\hat{\varsigma} \in \hat{\xi}} \hat{\varsigma}.$$

The new transfer function, $\hat{f}' : \hat{\Sigma} \to \hat{\Sigma}$, monotonically accumulates all of the visited states into a single state:

$$\hat{f}'(\hat{\varsigma}) = \hat{\varsigma} \sqcup \bigsqcup_{\hat{\varsigma} \Rightarrow \hat{\varsigma}'} \hat{\varsigma}'.$$

5.1 Complexity

This second abstraction simplifies the calculation of an upper bound on computational complexity. The structure of the set $\hat{\Sigma}$ is a pair of maps into sets:

$$\hat{\Sigma} = \left(\widehat{TID} \to \mathcal{P}(\widehat{Context})\right) \times \left(\widehat{Addr} \to \mathcal{P}(\widehat{Value})\right).$$

Each of these maps is, in effect, a table of bit vectors: the first with abstract thread ids on one axis and contexts on the other; and the second with abstract addresses on one axis and values on the other. The analysis monotonically flips bits on each pass. Thus, the maximum number of passes—the tallest ascending chain in the lattice $\hat{\Sigma}$—is:

$$|\widehat{\mathit{TID}}| \times |\widehat{\mathit{Context}}| + |\widehat{\mathit{Addr}}| \times |\widehat{\mathit{Value}}|.$$

Thus, the complexity of the analysis is determined by context-sensitivity, as with classical sequential flow analysis. For a standard monovariant analysis, the complexity is polynomial [17]. For a context-sensitive analysis with shared environments, the complexity is exponential [22]. For a context-sensitive analysis with flat environments, the complexity is again polynomial [15].

6 Related Work

This work traces its ancestry to Cousot and Cousot's work on abstract interpretation [3,4]. We could easily extend the fixed-point formulation with the implicit concretization function to arrive at an instance of traditional abstract interpretation. It is also a direct descendant of the line of work investigating control-flow in higher-programs that began with Jones [12] and Shivers [20,21].

The literature on static analysis of concurrency and higher-orderness is not empty, but it is spare. Much of it focuses on the special case of the analysis of futures. The work most notable and related to our own is that of Navabi and Jagannathan [16]. It takes Flanagan and Felleisen's notion of safe futures [7,8], and develops a dynamic and static analysis that can prevent a continuation from modifying a resource that one of its concurrent futures may modify. What makes this work most related to our own is that it is sound even in the presence of exceptions, which are, in essence, an upward-restricted form of continuations. Their work and our own own interact synergistically, since their safety analysis focuses on removing the parallel inefficiencies of safe futures; our flow analysis can remove the sequential inefficiencies of futures through the elimination of runtime type-checks. Yahav's work is the earliest to apply shape-analytic techniques to the analysis of concurrency [24].

It has taken substantial effort to bring the static analysis of higher-order programs to heel; to recite a few of the major challenges:

1. First-class functions from dynamically created closures over lambda terms create recursive dependencies between control- and data-flow; k-CFA co-analyzes control and data to factor and order these dependencies [21].
2. Environment-bearing closures over lambda terms impart fundamental intractabilities unto context-sensitive analysis [22]—intractabilities that were only recently side-stepped via flattened abstract environments [15].
3. The functional emphasis on recursion over iteration made achieving high precision difficult (or hopeless) without abstract garbage collection to recycle tail-call-bound parameters and continuations [14].

4. When closures keep multiple bindings to the same variable live, precise reasoning about side effects to these bindings requires the adaptation of shape-analytic techniques [11,13].
5. Precise reasoning about first-class continuations (and kin such as exceptions) required a harmful conversion to continuation-passing style until the advent of small-step abstraction interpretations for the pointer-refined CESK machine [23].

We see this work as another milestone on the path to robust static analysis of full-featured higher-order programs.

7 Limitations and Future Work

Since the shape of the store and the values within were not the primary focus of this work, it utilized a blunt abstraction. A compelling next step of this work would generalize the abstraction of the store relationally, so as to capture relations between the values at specific *addresses*. The key challenge in such an extension is the need to handle relations between abstract addresses which may represent *multiple* concrete addresses. Relations which universally quantify over the concrete constituents of an abstract address are a promising approach.

Acknowledgements. This material is based upon work supported by the NSF under Grant No. 1035658. The second author was supported by the National Science Foundation under Grant No. 0937060 to the Computing Research Association for the CIFellow Project. We also acknowledge support from the Park City Research Institute for Computer Science.

References

1. Balakrishnan, G., Reps, T.: Recency-abstraction for heap-allocated storage. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 221–239. Springer, Heidelberg (2006), http://dx.doi.org/10.1007/11823230_15, doi:10.1007/11823230_15
2. Chase, D.R., Wegman, M., Zadeck, F.K.: Analysis of pointers and structures. In: PLDI 1990: Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementationm, PLDI 1990, pp. 296–310. ACM, New York (1990)
3. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, pp. 238–252. ACM Press, New York (1977)
4. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL 1979: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL 1979, pp. 269–282. ACM Press, New York (1979)

5. Feeley, M.: An Efficient and General Implementation of Futures on Large Scale Shared-Memory Multiprocessors. PhD thesis, Brandeis University (April 1993)
6. Felleisen, M., Friedman, D.P.: Control operators, the SECD-machine, and the lambda-calculus. In: 3rd Working Conference on the Formal Description of Programming Concepts (August 1986)
7. Flanagan, C., Felleisen, M.: The semantics of future and its use in program optimization. In: POPL 1995: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 209–220. ACM, New York (1995)
8. Flanagan, C., Felleisen, M.: The semantics of future and an application. Journal of Functional Programming 9(01), 1–31 (1999)
9. Flanagan, C., Sabry, A., Duba, B.F., Felleisen, M.: The essence of compiling with continuations. In: PLDI 1993: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation, pp. 237–247. ACM, New York (1993)
10. Hudak, P.: A semantic model of reference counting and its abstraction. In: LFP 1986: Proceedings of the 1986 ACM Conference on LISP and Functional Programming, pp. 351–363. ACM, New York (1986)
11. Jagannathan, S., Thiemann, P., Weeks, S., Wright, A.: Single and loving it: must-alias analysis for higher-order languages. In: POPL 1998: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1998, pp. 329–341. ACM, New York (1998)
12. Jones, N.D.: Flow analysis of lambda expressions (preliminary version). In: Proceedings of the 8th Colloquium on Automata, Languages and Programming, pp. 114–128. Springer, London (1981)
13. Might, M.: Shape analysis in the absence of pointers and structure. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 263–278. Springer, Heidelberg (2010)
14. Might, M., Shivers, O.: Exploiting reachability and cardinality in higher-order flow analysis. Journal of Functional Programming 18(Special Double Issue 5-6), 821–864 (2008)
15. Might, M., Smaragdakis, Y., Van Horn, D.: Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis. In: PLDI 2010: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 305–315. ACM, New York (2010)
16. Navabi, A., Jagannathan, S.: Exceptionally safe futures. In: Field, J., Vasconcelos, V. (eds.) COORDINATION 2009. LNCS, vol. 5521, pp. 47–65. Springer, Heidelberg (2009)
17. Palsberg, J.: Closure analysis in constraint form. ACM Transactions on Programming Languages and Systems 17(1), 47–62 (1995)
18. Queinnec, C.: Continuations and web servers. Higher-Order and Symbolic Computation 17(4), 277–295 (2004)
19. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Transactions on Programming Languages and Systems 24(3), 217–298 (2002)
20. Shivers, O.: Control flow analysis in Scheme. In: PLDI 1988: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, pp. 164–174. ACM, New York (1988)

21. Shivers, O.G.: Control-Flow Analysis of Higher-Order Languages PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA (1991)
22. Van Horn, D., Mairson, H.G.: Deciding *k*CFA is complete for EXPTIME. In: ICFP 2008: Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming, pp. 275–282. ACM, New York (2008)
23. Van Horn, D., Might, M.: Abstracting abstract machines. In: ICFP 2010: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, pp. 51–62. ACM, New York (2010)
24. Yahav, E.: Verifying safety properties of concurrent java programs using 3-valued logic. In: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2001, pp. 27–40. ACM Press, New York (2001)

Abstract Domains of Affine Relations*

Matt Elder[1], Junghee Lim[1], Tushar Sharma[1], Tycho Andersen[1],
and Thomas Reps[1,2,**]

[1] University of Wisconsin, Madison, WI, USA
[2] GrammaTech, Inc., Ithaca, NY, USA

Abstract. This paper considers some known abstract domains for affine-relation analysis (ARA), along with several variants, and studies how they relate to each other. We show that the abstract domains of Müller-Olm/Seidl (MOS) and King/Søndergaard (KS) are, in general, incomparable, but give sound interconversion methods. We also show that the methods of King and Søndergaard can be applied without bit-blasting—while still using a bit-precise concrete semantics.

1 Introduction

The work reported in this paper was motivated by our work on TSL [16], which is a system for generating abstract interpreters for machine code. With TSL, one specifies an instruction set's concrete operational semantics by defining an interpreter

$$\texttt{interpInstr} : \texttt{instruction} \times \texttt{state} \rightarrow \texttt{state}.$$

For a given abstract domain $\mathcal{A}$, a sound abstract transformer for each instruction of the instruction set is obtained by defining a sound reinterpretation of each operation of the TSL meta-language as an operation over $\mathcal{A}$. By extending the reinterpretation to TSL expressions and functions—including `interpInstr`—the set of operator-level reinterpretations defines the desired set of abstract transformers for the instructions of the instruction set.

However, this method abstracts each TSL operation in isolation, and is therefore rather myopic. Moreover, the operators that TSL provides to specify an instruction set's concrete semantics include arithmetic, logical, and "bit-twiddling" operations. The latter include left-shift; arithmetic and logical right-shift; bitwise-and, bitwise-or, and bitwise-xor; etc. Few abstract domains retain precision over the full gamut of such operations.

A more global approach that considers the semantics of an entire instruction (or, even better, an entire basic block or other path fragment) can yield a more

* Supported, in part, by NSF under grants CCF-{0810053, 0904371}, by ONR under grants N00014-{09-1-0510, 10-M-0251}, by ARL under grant W911NF-09-1-0413, and by AFRL under grants FA9550-09-1-0279 and FA8650-10-C-7088. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring agencies.

** T. Reps has an ownership interest in GrammaTech, Inc., which has licensed elements of the technology discussed in this publication.

E. Yahav (Ed.): SAS 2011, LNCS 6887, pp. 198–215, 2011.

precise transformer. One way to specify the goals of such a global approach is through the notion of *symbolic abstraction* [22]:

- An abstract domain $\mathcal{A}$ is said to support a *symbolic implementation of the* α *function* of a Galois connection if, for every logical formula ψ that specifies (symbolically) a set of concrete stores $[\![\psi]\!]$, there is a method $\widetilde{\alpha}$ that finds a sound abstract value $\widetilde{\alpha}(\psi) \in \mathcal{A}$ that over-approximates $[\![\psi]\!]$. That is, $[\![\psi]\!] \subseteq \gamma(\widetilde{\alpha}(\psi))$, where $[\![\psi]\!]$ denotes the meaning function for the logic.
- For some abstract domains, it is even known how to perform a *best* symbolic implementation of α, denoted by $\widehat{\alpha}$ [22]. For every ψ, $\widehat{\alpha}$ finds the best value in $\mathcal{A}$ that over-approximates $[\![\psi]\!]$.

In particular, the issue of "myopia" is addressed by first creating a logical formula φ_I that captures the concrete semantics of each instruction I (or basic block, or path fragment) in quantifier-free bit-vector logic (QFBV), and then performing $\widetilde{\alpha}(\varphi_I)$ or $\widehat{\alpha}(\varphi_I)$. (The generation of a QFBV formula that, with no loss of precision, captures the concrete semantics of an instruction or basic block is a problem that itself fits the TSL operator-reinterpretation paradigm [16, §3.4].)

We explored how to address these issues using two existing abstract domains for affine-relation analysis (ARA)—one defined by Müller-Olm and Seidl (MOS) [19,21] and one defined by King and Søndergaard (KS) [11,12]—as well as a third domain of *affine generators* that we introduce. (Henceforth, the three domains are referred to as MOS, KS, and AG, respectively.) All three domains represent sets of points that satisfy affine relations over variables that hold machine integers, and are based on an extension of linear algebra to modules over a ring [8,7,1,25,19,21]. The contributions of our work can be summarized as follows:

- For MOS, it was not previously known how to perform $\widetilde{\alpha}_{\text{MOS}}(\varphi)$ in a nontrivial fashion (e.g., other than defining $\widetilde{\alpha}_{\text{MOS}}$ to be $\lambda f.\top$). In contrast, King and Søndergaard gave an algorithm for $\widehat{\alpha}_{\text{KS}}$ [12, Fig. 2], which led us to examine more closely how MOS and KS are related. A KS value consists of a set of *constraints* on the values of variables. We introduce a third abstract domain, AG, which can be considered to be the *generator* counterpart of KS. A KS constraint-based value can be converted to an AG value with no loss of precision, and vice versa.

  In contrast, we show that MOS and KS/AG are, in general, *incomparable*. However, we give sound interconversion methods: we show that an AG value v_{AG} can be converted to an over-approximating MOS value v_{MOS}—i.e., $\gamma(v_{\text{AG}}) \subseteq \gamma(v_{\text{MOS}})$—and that an MOS value w_{MOS} can be converted to an over-approximating AG value w_{AG}—i.e., $\gamma(w_{\text{MOS}}) \subseteq \gamma(w_{\text{AG}})$.

  Consequently, by means of the conversion path $\varphi \rightarrow \text{KS} \rightarrow \text{AG} \rightarrow \text{MOS}$, we show how to perform $\widetilde{\alpha}_{\text{MOS}}(\varphi)$ (§4.5).
- To apply the techniques described in the two King and Søndergaard papers [11,12], it is necessary to perform bit-blasting. Their goal is to create implementations of operations that are precise, modulo the inherent limitations of precision that stem from using KS. They use bit-blasting to express a

bit-precise concrete semantics for a statement or basic block. Working at the bit level lets them track the effect of non-linear bit-twiddling operations, such as shift and xor.

One drawback of bit-blasting is the huge number of variables that it introduces (e.g., 32 or 64 bit-valued variables for each `int`-valued program variable). Given that one is performing numerous cubic-time operations on the matrices that arise, there is a question as to whether the bit-blasted version of KS could ever be applied to problems of substantial size. The times reported by King and Søndergaard are quite high [12, §7], although they state that there is room for improvement by, e.g., using sparse matrices.

In our work, we use an SMT solver rather than a SAT solver, and show that implementations of operations that are best operations for the KS domain can be obtained without resorting to bit-blasting. Instead, we work with QFBV formulas that capture symbolically the precise bit-level semantics of each instruction or basic block, and take advantage of the ability of $\widehat{\alpha}_{\mathrm{KS}}$ to create best word-level transformers.[1]

The greatly reduced number of variables that comes from working at word level opens up the possibility of applying our methods to much larger problems, and in particular to performing interprocedural analysis. We show how to use the KS domain as the basis for interprocedural ARA. In particular, we use a two-vocabulary version of KS to create a weight domain for a weighted pushdown system (WPDS) [23,2,10] (§5).

In addition to the specific contributions listed above, this paper provides insight on the range of options one has for performing affine-relation analysis, and how the different approaches relate to each other.

Organization. The remainder of the paper is organized as follows: §2 summarizes relevant features of the various ARA domains considered in the paper. §3 presents the AG domain, and shows how an AG value can be converted to a KS value, and vice versa. §4 presents our results on the incomparability of the MOS and KS domains, but gives sound methods to convert a KS value into an over-approximating MOS value, and vice versa. §5 explains how to use the KS domain for interprocedural analysis without bit-blasting. §6 presents experimental results. §7 discusses related work. Proofs can be found in a companion technical report [4].

2 Terminology and Notation

All numeric values in this paper are integers in $\mathbb{Z}_{2^w}$ for some bit width w. That is, values are machine integers with the standard machine addition and multiplication. Addition and multiplication in $\mathbb{Z}_{2^w}$ form a ring, not a field, so some

[1] The two methods are not entirely comparable because the bit-blasting approach works with a great deal more variables (to represent the values of individual bits). However, for word-level properties the two are comparable. For instance, both can discover that the action of an xor-based swap is to exchange the values of two program variables.

facets of standard linear algebra do not apply (and thus we must regard our intuitions about linear algebra with caution). In particular, all odd elements in $\mathbb{Z}_{2^w}$ have a multiplicative inverse (which may be found in time $O(\log w)$ [26, Fig. 10-5]), but no even elements have a multiplicative inverse. The *rank* of a value $x \in \mathbb{Z}_{2^w}$ is the maximum integer $p \leq w$ such that $2^p | x$. For example, $\text{rank}(1) = 0$, $\text{rank}(12) = 2$, and $\text{rank}(0) = w$.

Throughout the paper, k is the size of the *vocabulary*, the variable set under analysis. A *two-vocabulary* relation is a relation between values of variables in its *pre-state* vocabulary to values of variables in its *post-state* vocabulary.

Matrix addition and multiplication are defined as usual, forming a matrix ring. We denote the transpose of a matrix M by M^t. A *one-vocabulary matrix* is a matrix with $k + 1$ columns. A *two-vocabulary matrix* is a matrix with $2k + 1$ columns. In each case, the "+1" is for technical reasons (which vary according to what kind of matrix we are dealing with). I denotes the (square) identity matrix (whose size can be inferred from context).

Actual states in the various abstract domains are represented by k-length row vectors. The *row space* of a matrix M is $\text{row}\, M \overset{\text{def}}{=} \{x \mid \exists w\colon wM = x\}$. When we speak of the "null space" of a matrix, we actually mean the set of row vectors whose transposes are in the traditional null space of the matrix. Thus, we define $\text{null}^t\, M \overset{\text{def}}{=} \{x \mid Mx^t = 0\}$.

Matrices in Howell Form. An appreciation of how linear algebra in rings differs from linear algebra in fields can be obtained by seeing how certain issues are finessed when converting a matrix to *Howell form* [8]. The Howell form of a matrix is an extension of reduced row-echelon form [17] suitable for matrices over $\mathbb{Z}_n$. Because Howell form is a canonical form for matrices over principal ideal rings [8,25], it provides a way to test pairs of abstract-domain elements for equality of their concretizations—an operation needed by analysis algorithms to determine when a fixed point is reached.

Definition 1. *The leftmost nonzero value in a row vector is its* **leading value***, and the leading value's index is the* **leading index***. A matrix M is in* **row-echelon form** *iff*

- *All rows consisting entirely of zeroes are at the bottom.*
- *The sequence of leading indices of rows is strictly increasing.*

If M is in row-echelon form, let $[M]_i$ denote the matrix that consists of all rows of M whose leading index is i or greater.

A matrix M is in **Howell form** *iff*

1. *M is in row-echelon form,*
2. *the leading value of every row is a power of two,*
3. *each leading value is the largest value in its column, and*
4. *for every row r of M, for any $p \in \mathbb{Z}$, if i is the leading index of $2^p r$, then $2^p r \in \text{row}[M]_i$.*

Suppose that $w = 4$. Item 4 of Defn. 1 is illustrated by $M = \left[\begin{smallmatrix} 4 & 2 & 4 \\ 0 & 4 & 0 \end{smallmatrix}\right]$. The first row of M has leading index 1. Multiplying the first row by 4 produces $[0\ 8\ 0]$,

Algorithm 1. Howellize: Put the matrix G in Howell form.

```
 1: procedure Howellize($G$)
 2:     Let $j = 0$                                   ▷ $j$ is the number of already-Howellized rows
 3:     for all $i$ from 1 to $2k + 1$ do
 4:         Let $R = \{$all rows of $G$ with leading index $i\}$
 5:         if $R \neq \emptyset$ then
 6:             Pick an $r \in R$ that minimizes rank $r_i$
 7:             Pick the odd $u$ and rank $p$ so that $u2^p = r_i$
 8:             $r \leftarrow u^{-1} r$                          ▷ Adjust $r$, leaving $r_i = 2^p$
 9:             for all $s$ in $R \setminus \{r\}$ do
10:                 Pick the odd $v$ and rank $t$ so that $v2^t = s_i$
11:                 $s \leftarrow s - \left(v2^{t-p}\right) r$                   ▷ Zero out $s_i$
12:                 if row $s$ contains only zeros then
13:                     Remove $s$ from $G$
14:             In $G$, swap $r$ with $G_{j+1}$              ▷ Place $r$ for row-echelon form
15:             for all $h$ from 1 to $j$ do          ▷ Set values above $r_i$ to be $0 \leq \cdot < r_i$
16:                 $d \leftarrow G_{h,i} \gg p$                 ▷ Pick $d$ so that $0 \leq G_{h,i} - dr_i < r_i$
17:                 $G_h \leftarrow G_h - dr$               ▷ Adjust row $G_h$, leaving $0 \leq G_{h,i} < r_i$
18:             if $r_i \neq 1$ then                  ▷ Add logical consequences of $r$ to $G$
19:                 Add $2^{w-p} r$ as last row of $G$        ▷ New row has leading index $> i$
20:             $j \leftarrow j + 1$
```

which has leading index 2. This meets condition 4 because $[0\ 8\ 0] = 2 \cdot [0\ 4\ 0]$, so $[0\ 8\ 0] \in \text{row}[M]_2$.

The Howell form of a matrix is unique among all matrices with the same row space (or null space) [8]. As mentioned above, Howell form provides a way to test pairs of KS or AG elements for equality of their concretizations.

The notion of a *saturated set of generators* used by Müller-Olm and Seidl [21] is closely related to Howell form, but is defined for an unordered set of generators rather than row-vectors arranged in a matrix, and has no analogue of item 3. The algorithms of Müller-Olm and Seidl do not compute multiplicative inverses (see §7), so a saturated set has no analogue of item 2. Consequently, a saturated set is not canonical among generators of the same space.

Our technique for putting a matrix in Howell form is given as procedure Howellize (Alg. 1). Much of Howellize is similar to a standard Gaussian-elimination algorithm, and it has the same overall cubic-time complexity as Gaussian elimination. In particular, Howellize minus lines 15–19 puts G in row-echelon form (item 1 of Defn. 1) with the leading value of every row a power of two. (Line 8 enforces item 2 of Defn. 1.) Howellize differs from standard Gaussian elimination in how the pivot is picked (line 6) and in how the pivot is used to zero out other elements in its column (lines 7–13). Lines 15–17 of Howellize enforce item 3 of Defn. 1, and lines 18–19 enforce item 4. Lines 12–13 remove all-zero rows, which is needed for Howell form to be canonical.

The Affine Generator Domain. An element in the Affine Generator domain (AG) is a two-vocabulary matrix whose rows are the affine generators of a two-vocabulary relation.

An AG element is an r-by-$(2k+1)$ matrix G, with $0 < r \leq 2k+1$. The concretization of an AG element is

$$\gamma_{\mathrm{AG}}(G) \overset{\mathrm{def}}{=} \left\{(x, x') \mid x, x' \in \mathbb{Z}_{2^w}^k \wedge [1|x\ x'] \in \operatorname{row} G\right\}.$$

The AG domain captures all two-vocabulary affine spaces, and treats them as relations between pre-states and post-states.

The bottom element of the AG domain is the empty matrix, and the AG element that represents the identity relation is the matrix $\left[\begin{array}{c|cc} 1 & 0 & 0 \\ \hline 0 & I & I \end{array}\right]$. To compute the join of two AG matrices, stack the two matrices vertically and Howellize the result.

The King/Søndergaard Domain. An element in the King/Søndergaard domain (KS) is a two-vocabulary matrix whose rows represent constraints on a two-vocabulary relation. A KS element is an r-by-$(2k+1)$ matrix X, with $0 \leq r \leq 2k+1$. The concretization of a KS element is

$$\gamma_{\mathrm{KS}}(X) \overset{\mathrm{def}}{=} \left\{(x, x') \mid x, x' \in \mathbb{Z}_{2^w}^k \wedge [x\ x'|1] \in \operatorname{null}^t G\right\}.$$

Like the AG domain, the KS domain captures all two-vocabulary affine spaces, and treats them as relations between pre-states and post-states. The original KS paper [11] gives polynomial-time algorithms for join and projection; projection can be used to implement composition.

It is easy to read off affine relations from a KS element: if $[a_1 \cdots a_k\ a'_1 \cdots a'_k\ |{-b}]$ is a row of X, then $\sum_i a_i x_i + \sum_i a'_i x'_i = b$ is a constraint on $\gamma_{\mathrm{KS}}(X)$. The conjunction of these constraints describes $\gamma_{\mathrm{KS}}(X)$ precisely.

The bottom element of the KS domain is the matrix $[0\ 0|1]$, and the KS element that represents the identity relation is the matrix $[I\ \text{-}I|0]$.

A Howell-form KS element can easily be checked for emptiness: it is empty if and only if it contains a row whose leading entry is in its last column. In that sense, an implementation of the KS domain in which all elements are kept in Howell form has redundant representations of bottom (whose concretization is $\emptyset$). However, such KS elements can always be detected during Howellize and replaced by the canonical representation of bottom, $[0\ 0|1]$.

The Müller-Olm/Seidl Domain. An element in the Müller-Olm/Seidl domain (MOS) is an affine set of affine transformers, as detailed in [21]. An MOS element represents a set of $(k+1)$-by-$(k+1)$ matrices. Each matrix T is a one-vocabulary transformer of the form $T = \left[\begin{array}{c|c} 1 & b \\ \hline 0 & M \end{array}\right]$, which represents the state transformation $x' := x \cdot M + b$, or, equivalently, $[1|x'] := [1|x]\,T$.

An MOS element $\mathcal{B}$ consists of a set of $(k+1)$-by-$(k+1)$ matrices, and represents the affine span of the set, denoted by $\langle \mathcal{B} \rangle$ and defined as follows:

$$\langle \mathcal{B} \rangle \overset{\mathrm{def}}{=} \left\{ T \;\middle|\; \exists w \in \mathbb{Z}_{2^w}^{|\mathcal{B}|} : T = \sum_{B \in \mathcal{B}} w_B B \wedge T_{1,1} = 1 \right\}.$$

The meaning of $\mathcal{B}$ is the union of the graphs of the affine transformers in $\langle B \rangle$

$$\gamma_{\text{MOS}}\left(\mathcal{B}\right) \overset{\text{def}}{=} \left\{(x, x') \,\middle|\, x, x' \in \mathbb{Z}_{2^w}^k \wedge \exists T \in \langle \mathcal{B} \rangle \colon [1|x]\, T = [1|x'] \right\}.$$

The bottom element of the MOS domain is $\emptyset$, and the MOS element that represents the identity relation is the singleton set $\{I\}$.

The operations join and compose can be performed in polynomial time. If $\mathcal{B}$ and $\mathcal{C}$ are MOS elements, then $\mathcal{B} \sqcup \mathcal{C} = \textsc{Howellize}\,(\mathcal{B} \cup \mathcal{C})$ and $\mathcal{B} \circ \mathcal{C} = \textsc{Howellize}\,\{BC \,|\, B \in \mathcal{B} \wedge C \in \mathcal{C}\}$. In this setting, $\textsc{Howellize}$ of a set of $(k+1)$-by-$(k+1)$ matrices $\{M_1, \ldots, M_n\}$ means "Apply Alg. 1 to a larger, n-by-$(k+1)^2$ matrix, each of whose rows is the linearization (e.g., in row-major order) of one of the M_i."

3 Relating AG and KS Elements

AG and KS are equivalent domains. One can convert an AG element to an equivalent KS element with no loss of precision, and vice versa. In essence, these are a single abstract domain with two representations: constraint form (KS) and generator form (AG).

We use an operation similar to singular value decomposition, called diagonal decomposition:

Definition 2. *The* **diagonal decomposition** *of a square matrix M is a triple of matrices, L, D, R, such that $M = LDR$; L and R are invertible matrices; and D is a diagonal matrix in which all entries are either 0 or a power of 2.*

Müller-Olm and Seidl give a decomposition algorithm that nearly performs diagonal decomposition [21, Lemma 2.9], except that the entries in their D might not be powers of 2. We can easily adapt that algorithm. Suppose that their method yields LDR (where L and R are invertible). Pick u and r so that $u_i 2^{r_i} = D_{i,i}$ with each u_i odd, and define U as the diagonal matrix where $U_{i,i} = u_i$. (If $D_{i,i} = 0$, then $u_i = 1$.) It is easy to show that U is invertible. Let $L' = LU$ and $D' = U^{-1}D$. Consequently, $L'D'R = LDR = M$, and $L'D'R$ is a diagonal decomposition.

From diagonal decomposition we derive the dual operation, denoted by $\cdot^{\perp}$, such that the rows of $M^{\perp}$ generate the null space of M, and vice versa.

Definition 3. *The* **dualization** *of M is $M^{\perp}$, where:*

- $\textsc{Pad}(M)$ *is the* $(2k+1)$*-by-*$(2k+1)$ *matrix* $\left[\begin{smallmatrix} M \\ \mathbf{0} \end{smallmatrix}\right]$,
- L, D, R *is the diagonal decomposition of* $\textsc{Pad}(M)$,
- T *is the diagonal matrix with* $T_{i,i} \overset{\text{def}}{=} 2^{w - \operatorname{rank}(D_{i,i})}$, *and*
- $M^{\perp} \overset{\text{def}}{=} \left(L^{-1}\right)^t T \left(R^{-1}\right)^t$

This definition of dualization has the following useful property:

Theorem 1. *For any matrix M,* $\operatorname{null}^t M = \operatorname{row} M^{\perp}$ *and* $\operatorname{row} M = \operatorname{null}^t M^{\perp}$.

We can therefore use dualization to convert between equivalent KS and AG elements. For a given (padded, square) AG matrix $G = [c|Y\ Y']$, we seek a KS matrix Z of the form $[X\ X'|b]$ such that $\gamma_{\text{KS}}(Z) = \gamma_{\text{AG}}(G)$. We construct Z by letting $[b|X\ X'] = G^{\perp}$ and permuting those columns to $Z \overset{\text{def}}{=} [X\ X'|b]$. This works by Thm. 1, and because

$$\begin{aligned}\gamma_{\text{AG}}(G) &= \{(x, x') \mid [1|x\ x'] \in \operatorname{row} G\} \\ &= \left\{(x, x') \,\middle|\, [1|x\ x'] \in \operatorname{null}^t G^{\perp}\right\} \\ &= \left\{(x, x') \,\middle|\, [x\ x'|1] \in \operatorname{null}^t Z\right\} = \gamma_{\text{KS}}(Z)\,.\end{aligned}$$

Furthermore, to convert from any KS matrix to an equivalent AG matrix, we reverse the process. Reversal is possible because dualization is an involution: for any matrix M, $\left(M^{\perp}\right)^{\perp} = M$.

4 Relating KS and MOS

4.1 MOS and KS are Incomparable

The MOS and KS domains are incomparable: some relations are expressible in each domain that are not expressible in the other. Intuitively, the central difference is that MOS is a domain of sets of *functions*, while KS is a domain of *relations*.

KS can capture restrictions on both the pre-state and post-state vocabularies while MOS captures restrictions only on its post-state vocabulary. For example, when $k = 1$, the KS element for "assume $x = 2$" is $\left\{\left[\begin{array}{cc|c} 1 & 0 & -2 \\ 1 & -1 & 0 \end{array}\right]\right\}$, i.e., "$x = 2 \wedge x' = x$". An MOS element cannot encode an assume statement. For "assume $x = 2$", the best MOS element is the identity transformer $\left\{\left[\begin{array}{c|c} 1 & 0 \\ \hline 0 & 1 \end{array}\right]\right\}$. In general, an MOS element cannot encode a non-trivial condition on the pre-state. If an MOS element contains a single transition, it encodes that transition for every possible pre-state. Therefore, KS can encode relations that MOS cannot encode.

On the other hand, an MOS element can encode two-vocabulary relations that are not affine. One example is the matrix basis $\mathcal{B} = \left\{\left[\begin{array}{c|cc} 1 & 0 & 0 \\ \hline 0 & 1 & 1 \\ 0 & 0 & 0 \end{array}\right], \left[\begin{array}{c|cc} 1 & 0 & 0 \\ \hline 0 & 0 & 0 \\ 0 & 1 & 1 \end{array}\right]\right\}$. The set that $\mathcal{B}$ encodes is

$$\begin{aligned}\gamma_{\text{MOS}}(\mathcal{B}) &= \left\{ \left[x\ y\ x'\ y'\right] \,\middle|\, \begin{array}{c} \exists w_0, w_1 \colon \left[\,1\middle|x\ y\,\right] \left[\begin{array}{c|cc} 1 & 0 & 0 \\ \hline 0 & w_0 & w_0 \\ 0 & w_1 & w_1 \end{array}\right] = \left[\,1\middle|x'\ y'\,\right] \\ \wedge\ w_0 + w_1 = 1 \end{array} \right\} \\ &= \left\{\left[x\ y\ x'\ y'\right] \,\middle|\, \exists w_0 \colon x' = y' = w_0 x + (1 - w_0) y\right\} \\ &= \left\{\left[x\ y\ x'\ y'\right] \,\middle|\, \exists w_0 \colon x' = y' = x + (1 - w_0)(y - x)\right\} \\ &= \left\{\left[x\ y\ x'\ y'\right] \,\middle|\, \exists p \colon x' = y' = x + p(y - x)\right\} \qquad (1)\end{aligned}$$

Affine spaces are closed under affine combinations of their elements. Thus, $\gamma_{\text{MOS}}(\mathcal{B})$ is not an affine space because some affine combinations of its elements are not in $\gamma_{\text{MOS}}(\mathcal{B})$. For instance, let $a = \begin{bmatrix}1 & -1 & 1 & 1\end{bmatrix}$, $b = \begin{bmatrix}2 & -2 & 6 & 6\end{bmatrix}$, and $c = \begin{bmatrix}0 & 0 & -4 & -4\end{bmatrix}$. By Eqn. (1), we have $a \in \gamma_{\text{MOS}}(\mathcal{B})$ when $p = 0$ in Eqn. (1), $b \in \gamma_{\text{MOS}}(\mathcal{B})$ when $p = -1$, and $c \notin \gamma_{\text{MOS}}(\mathcal{B})$ (the equation "$-4 = 0 + p(0-0)$" has no solution for p). Moreover, $2a - b = c$, so c is an affine combination of a and b. Thus, $\gamma_{\text{MOS}}(\mathcal{B})$ is not closed under affine combinations of its elements, and so $\gamma_{\text{MOS}}(\mathcal{B})$ is not an affine space. Because every KS element encodes a two-vocabulary affine space, MOS can represent $\gamma_{\text{MOS}}(\mathcal{B})$ but KS cannot.

4.2 Converting MOS Elements to KS

Soundly converting an MOS element to a KS element is equivalent to stating two-vocabulary affine constraints satisfied by that MOS element. To convert an MOS element $\mathcal{B}$ to a KS element, we

1. build a two-vocabulary AG matrix from each one-vocabulary matrix in $\mathcal{B}$,
2. compute the join of all the AG matrices from Step 1, and
3. convert the resulting AG matrix to a KS element.

For Step 1, assume that $\mathcal{B} = \left\{ \left[\begin{array}{c|c} 1 & c_i \\ \hline 0 & N_i \end{array}\right] \,\middle|\, 0 < i \right\}$, $c_i \in \mathbb{Z}_{2^w}^{1\times k}$, and $N_i \in \mathbb{Z}_{2^w}^{k\times k}$. If the original MOS element $\mathcal{B}_0$ fails to satisfy this property, let $\mathcal{C} = \textsc{Basis}(\mathcal{B}_0)$, pick the unique $B \in \mathcal{C}$ such that $B_{1,1} = 1$, and let $\mathcal{B} = \{B\} \cup \{B + C \mid C \in \mathcal{C} \setminus \{B\}\}$. $\mathcal{B}$ now satisfies the property, and $\langle \mathcal{B} \rangle = \langle \mathcal{B}_0 \rangle$.

From $\mathcal{B}$, we construct the matrices $G_i = \left[\begin{array}{c|cc} 1 & 0 & c_i \\ \hline 0 & I & N_i \end{array}\right]$. Note that, for each matrix $B_i \in \mathcal{B}$ with corresponding matrix G_i, $\gamma_{\text{MOS}}(\{B_i\}) = \gamma_{\text{AG}}(G_i)$. In Step 2, we join the G_i matrices in the AG domain to yield one matrix G. Thm. 2 proves the soundness of this transformation from MOS to AG, i.e., $\gamma_{\text{AG}}(G) \supseteq \gamma_{\text{MOS}}(\mathcal{B})$. Finally, G is converted in Step 3 to an equivalent KS element by the method given in §3.

Theorem 2. *Suppose that $\mathcal{B}$ is an MOS element such that, for every $B \in \mathcal{B}$, $B = \left[\begin{array}{c|c} 1 & c_B \\ \hline 0 & M_B \end{array}\right]$ for some $c_B \in \mathbb{Z}_{2^w}^{1\times k}$ and $M_B \in \mathbb{Z}_{2^w}^{k\times k}$. Define $G_B = \left[\begin{array}{c|cc} 1 & 0 & c_B \\ \hline 0 & I & M_B \end{array}\right]$ and $G = \bigsqcup_{AG} \{G_B \mid B \in \mathcal{B}\}$. Then, $\gamma_{\textit{MOS}}(\mathcal{B}) \subseteq \gamma_{\textit{AG}}(G)$.*

4.3 Converting KS without Pre-State Guards to MOS

If a KS element is total with respect to pre-state inputs, then we can convert it to an equivalent MOS element. First, convert the KS element to an AG element G. When G expresses no restrictions on its pre-state, it has the form $G = \left[\begin{array}{c|cc} 1 & 0 & b \\ \hline 0 & I & M \\ 0 & 0 & R \end{array}\right]$, where $b \in \mathbb{Z}_{2^w}^{1\times k}$; $I, M \in \mathbb{Z}_{2^w}^{k\times k}$; and $R \in \mathbb{Z}_{2^w}^{k\times r}$ with $0 \le r \le k$.

Definition 4. *An AG matrix of the form $\left[\begin{array}{c|cc} 1 & 0 & b \\ \hline 0 & I & M \end{array}\right]$, such as the G_i matrices discussed in §4.2, is said to be in* **explicit form** *because it represents the state transformation $x' := x \cdot M + b$.*

Explicit form is desirable because we can read off the MOS element $\left\{ \left[\begin{array}{c|c} 1 & b \\ \hline 0 & M \end{array}\right] \right\}$ from the AG matrix of Defn. 4.

Algorithm 2. MakeExplicit: Transform an AG matrix G in Howell form to near-explicit form.

```
Require: $G$ is an AG matrix in Howell form
 1: procedure MakeExplicit($G$)
 2:     for all $i$ from 2 to $k+1$ do            ▷ Consider each col. of the pre-state voc.
 3:         if there is a row $r$ of $G$ with leading index $i$ then
 4:             if rank $r_i > 0$ then
 5:                 for all $j$ from 1 to $2k+1$ do        ▷ Build $s$ from $r$, with $s_i = 1$
 6:                     $s_j \leftarrow r_j \gg \operatorname{rank} r_i$
 7:                 Append $s$ to $G$
 8:                 $G \leftarrow$ Howellize($G$)
 9:     for all $i$ from 2 to $k+1$ do
10:         if there is no row $r$ of $G$ with leading index $i$ then
11:             Insert, as the $i^{\text{th}}$ row of $G$, a new row of all zeroes
```

G is not in explicit form because of the rows $[0|0\ R]$; however, G is quite close to being in explicit form, and we can read off a *set* of matrices to create an appropriate MOS element. We produce this set of matrices via the Shatter operation, where

$$\textsc{Shatter}(G) \overset{\text{def}}{=} \left\{ \left[\begin{array}{c|c} 1 & b \\ \hline 0 & M \end{array}\right] \right\} \cup \left\{ \left[\begin{array}{c|c} 0 & R_{j,*} \\ \hline 0 & 0 \end{array}\right] \;\middle|\; 0 < j \le r \right\}, \text{ where } R_{j,*} \text{ is row } j \text{ of } R.$$

As shown in Thm. 3, $\gamma_{\text{AG}}(G) = \gamma_{\text{MOS}}(\textsc{Shatter}(G))$. Intuitively, this holds because coefficients in an affine combination of the rows of G correspond cleanly to coefficients in an affine combination of the $R_{j,*}$ matrices in $\textsc{Shatter}(G)$.

Theorem 3. *When* $G = \left[\begin{array}{c|cc} 1 & 0 & b \\ \hline 0 & I & M \\ 0 & 0 & R \end{array}\right]$, *then* $\gamma_{AG}(G) = \gamma_{MOS}(\textsc{Shatter}(G))$.

4.4 Converting KS with Pre-State Guards to MOS

If a KS element is not total with respect to pre-state inputs, then there is no MOS element with the same concretization. However, we can find sound over-approximations within MOS for such KS elements.

We convert the KS element into an AG matrix G as in §4.3 and put G in Howell form. There are two ways that G can enforce guards on the pre-state vocabulary: it might contain one or more rows whose leading value is even, or it might skip some leading indexes in row-echelon form.

While we cannot put G in explicit form, we can run MakeExplicit to coarsen G so that it is close enough to the form that arose in §4.3. Adding extra rows to an AG element can only enlarge its concretization. Thus, to handle a leading value $2^p, p > 0$ in the pre-state vocabulary, MakeExplicit introduces an extra, over-approximating row constructed by copying the row with leading value 2^p and right-shifting each value in the copied row by p bits (lines 4–8). After the loop on lines 2–8 finishes, every leading value in a row that generates pre-state-vocabulary values is 1. MakeExplicit then introduces all-zero rows so that each leading element from the pre-state vocabulary lies on the diagonal (lines 9–11).

Example 1. Suppose that $k = 3$, $w = 4$, and $G = \left[\begin{array}{c|cccccc} 1 & 0 & 2 & 0 & 0 & 0 & 0 \\ \hline & 4 & 0 & 12 & 2 & 4 & 0 \\ & & & & 4 & 0 & 8 \end{array}\right]$. After line 11 of MakeExplicit, all pre-state vocabulary leading values of G have been made ones, and the resulting G' has $\operatorname{row} G' \supseteq \operatorname{row} G$. In our case, $G' = \left[\begin{array}{c|cccccc} 1 & 0 & 2 & 0 & 0 & 0 & 0 \\ \hline & 1 & 0 & 3 & 0 & 1 & 0 \\ & & & & 2 & 0 & 0 \\ & & & & & & 8 \end{array}\right]$. To handle "skipped" indexes, lines 9–11 insert all-zero rows into G' so that each leading element from the pre-state vocabulary lies on the diagonal. The resulting matrix is $\left[\begin{array}{c|cccccc} 1 & 0 & 2 & 0 & 0 & 0 & 0 \\ \hline & 1 & 0 & 3 & 0 & 1 & 0 \\ & & 0 & 0 & 0 & 0 & 0 \\ & & & 0 & 0 & 0 & 0 \\ & & & & 2 & 0 & 0 \\ & & & & & & 8 \end{array}\right]$.

Theorem 4. *For any* G, $\gamma_{AG}(G) \subseteq \gamma_{MOS}(\textsc{Shatter}(\textsc{MakeExplicit}(G)))$.

Thus, we can use Shatter, MakeExplicit, and the AG–to–KS conversion of §3 to obtain an over-approximation of a KS element in MOS.

Example 2. The final MOS value for Ex. 1 is $\left\{ \left[\begin{array}{c|ccc} 1 & 0 & 0 & 0 \\ \hline 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right], \left[\begin{array}{c|ccc} 0 & 2 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right], \left[\begin{array}{c|ccc} 0 & 0 & 0 & 8 \\ \hline 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right] \right\}$.

4.5 Symbolic Implementation of the α Function for MOS

As mentioned in the Introduction, it was not previously known how to perform symbolic abstraction for MOS. Using $\widehat{\alpha}_{\text{KS}}$ [12, Fig. 2] in conjunction with the algorithms from §3 and §4.4, we can soundly define $\widetilde{\alpha}_{\text{MOS}}(\varphi) \overset{\text{def}}{=}$ **let** $G =$ $\textsc{ConvertKStoAG}\,(\widehat{\alpha}_{\text{KS}}(\varphi))$ **in** $\textsc{Shatter}\,(\textsc{MakeExplicit}\,(G))$.

5 Using KS for Interprocedural Analysis

This section describes how to use the KS domain in interprocedural-analysis algorithms in the style of Sharir and Pnueli [24], Knoop and Steffen [13], Müller-Olm and Seidl [18], and Lal et al. [15].

Project. In [11, §3], King and Søndergaard describe a way to project a KS element X onto a suffix $x_i, \ldots, x_k$ of its vocabulary: (i) put X in row-echelon form, and (ii) remove every row a in which any of $a_1, \ldots, a_{i-1}$ is nonzero. However, when the leading values of X are not all 1, step (ii) is not guaranteed to produce the most-precise projection of X onto $x_i, \ldots, x_k$ (although the value obtained is always sound). Instead, we put X in Howell form, and by Thm. 5, step (ii) returns the most-precise projection.

Theorem 5. *Suppose that* M *has* c *columns. If matrix* M *is in Howell form,* $x \in \operatorname{null}^t M$ *if and only if* $\forall i \colon \forall y_1, \ldots y_{i-1} \colon \begin{bmatrix} y_1 & \cdots y_{i-1} & x_i & \cdots & x_c \end{bmatrix} \in \operatorname{null}^t [M]_i$.

Example 3. Suppose that $X = [4\ 2|6]$, with $w = 4$, and the goal is to project away the first column (for x_1). King and Søndergaard obtain the empty matrix, which represents no constraints on x_2. The Howell form of X is $\left[\begin{array}{cc|c} 4 & 2 & 6 \\ 0 & 8 & 8 \end{array}\right]$, and thus the most precise projection of X onto x_2 is $[8|8]$, which represents $x_2 \in \{1, 3, \ldots, 15\}$.

Compose. In [12, §5.2], King and Søndergaard present a technique to compose two-vocabulary affine relations. For completeness, that algorithm follows. Suppose that we have KS elements $Y = \left[Y_{\text{pre}}\ Y_{\text{post}} \middle| y\right]$ and $Z = \left[Z_{\text{pre}}\ Z_{\text{post}} \middle| z\right]$, where Y_* and Z_* are k-column matrices, and y and z are column vectors. We want to compute $Y \circ Z$; i.e., some X such that $(x, x'') \in \gamma_{\text{KS}}(X)$ if and only if $\exists x' \colon (x, x') \in \gamma_{\text{KS}}(Y) \wedge (x', x'') \in \gamma_{\text{KS}}(Z)$.

Because the KS domain has a projection operation, we can create $Y \circ Z$ by first constructing the three-vocabulary matrix $W = \left[\begin{array}{ccc|c} Y_{\text{post}} & Y_{\text{pre}} & 0 & y \\ Z_{\text{pre}} & 0 & Z_{\text{post}} & z \end{array}\right]$. Any element $(x', x, x'') \in \gamma_{\text{KS}}(W)$ has $(x, x') \in \gamma_{\text{KS}}(Y)$ and $(x', x'') \in \gamma_{\text{KS}}(Z)$. Consequently, projecting away the first vocabulary of W yields a matrix X such that $\gamma_{\text{KS}}(X) = \gamma_{\text{KS}}(Y) \circ \gamma_{\text{KS}}(Z)$, as required.

Join. In [11, §3], King and Søndergaard give a method to compute the join of two KS elements by building a $(6k+3)$-column matrix and projecting onto its last $2k+1$ variables. We improve their approach slightly, building a $(4k+2)$-column matrix and then projecting onto its last $2k+1$ variables. That is, to join two KS elements $Y = \left[Y_{\text{pre}}\ Y_{\text{post}} \middle| y\right]$ and $Z = \left[Z_{\text{pre}}\ Z_{\text{post}} \middle| z\right]$, we first construct the matrix $\left[\begin{array}{ccccc|c} -Y_{\text{pre}} & -Y_{\text{post}} & -y & Y_{\text{pre}} & Y_{\text{post}} & y \\ Z_{\text{pre}} & Z_{\text{post}} & z & 0 & 0 & 0 \end{array}\right]$, and then project onto the last $2k+1$ columns. Roughly speaking, this works because $\begin{bmatrix} -Y & Y \\ Z & 0 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = 0$ if and only if $Y(v-u) = 0 \wedge Zu = 0$.

Because $(v-u) \in \operatorname{null} Y$, and $u \in \operatorname{null} Z$, $v = ((v-u)+u) \in \operatorname{null} Y + \operatorname{null} Z$. The correctness of this join algorithm can be proved by starting from the King and Søndergaard join matrix [11, §3], applying row-reductions, permuting columns due to be projected away, and partially performing the projection.

Merge Functions. Knoop and Steffen [13] extended the Sharir and Pnueli algorithm [24] for interprocedural dataflow analysis to handle local variables. At a call site at which procedure P calls procedure Q, the local variables of P are modeled as if the current incarnations of P's locals are stored in locations that are inaccessible to Q and to procedures transitively called by Q. Because the contents of P's locals cannot be affected by the call to Q, a *merge function* is used to combine them with the value returned by Q to create the state after Q returns. Other work using merge functions includes Müller-Olm and Seidl [18] and Lal et al. [15].

To simplify the discussion, assume that all scopes have the same number of locals L. Each merge function is of the form

$$\textsc{Merge}(a, b) \stackrel{\text{def}}{=} \textsc{ReplaceLocals}(b) \circ a.$$

Suppose that vocabulary i consists of sub-vocabularies g_i and l_i. The operation ReplaceLocals(b) is defined as follows:

1. Project away vocabulary l_2 from b.
2. Insert L columns for l_2 in which all entries are 0.
3. Append L rows, $[0, I, 0, -I|0]$, so that in ReplaceLocals(b) each variable in vocabulary l_2 is constrained to have the value of the corresponding variable in vocabulary l_1.

The $\widehat{\alpha}$ Operation. King and Søndergaard give an algorithm for $\widehat{\alpha}$ [12, Fig. 2]. That algorithm needs the minor correction of using Howell form instead of row-echelon form for the projections that take place in its join operations, as discussed above.

6 Experiments

Our experiments were run on a single core of a single-processor quad-core 3.0 GHz Xeon computer running 64-bit Windows XP (Service Pack 2), configured so that a user process has 4 GB of memory. To implement $\widehat{\alpha}_{\mathrm{KS}}$, we used the Yices solver [3], with the timeout for each invocation set to 3 seconds. The experiments were designed to answer the following questions:

1. Is it faster to use MOS or KS?
2. Does MOS or KS yield more precise answers? This question actually has several versions, depending on whether we are interested in
   - the two-vocabulary transformers for individual statements (or basic blocks)
   - the one-vocabulary affine relations that hold at program points

We ran each experiment on x86 machine code, computing affine relations over the x86 registers.

To address question 1, we ran ARA on a corpus of Windows utilities using the WALi [10] system for weighted pushdown systems (WPDSs) [23,2]. We used two weight domains: (i) a weight domain of TSL-generated MOS transformers, and (ii) a weight domain of $\widehat{\alpha}_{\mathrm{KS}}$-generated KS transformers. The weight on each WPDS rule encoded the MOS/KS transformer for a basic block $B = [I_1, \ldots, I_k]$ of the program, including a jump or branch to a successor block.

- In the case of MOS, the semantic specification of each instruction $I_j \in B$ is evaluated according to the MOS reinterpretation of the operations of the TSL meta-language to obtain $[\![I_j]\!]_{\mathrm{MOS}}$. ($[\![\cdot]\!]_{\mathrm{MOS}}$ denotes the MOS semantics for an instruction.) The single-instruction transformers are then composed: $w^B_{\mathrm{MOS}} := [\![I_k]\!]_{\mathrm{MOS}} \circ \ldots \circ [\![I_1]\!]_{\mathrm{MOS}}$.
- In the case of KS, a formula φ_B is created that captures the concrete semantics of B, and then the KS weight for B is obtained by performing $w^B_{\mathrm{KS}} := \widehat{\alpha}_{\mathrm{KS}}(\varphi_{\mathcal{B}})$.

Prog.	Measures of size				Performance (x86)							Better KS
					MOS			KS				precision
name	instrs	CFGs	BBs	branches	WPDS	post*	query	WPDS	t/o	post*	query	1-voc.
finger	532	18	298	48	0.969	0.281	0.094	121.0	5	0.297	0.016	11/48 (23%)
subst	1093	16	609	74	1.422	0.266	0.031	199.0	4	0.328	0.094	13/74 (18%)
label	1167	16	573	103	1.359	0.282	0.046	154.6	2	0.375	0.032	50/103 (49%)
chkdsk	1468	18	787	119	1.797	0.172	0.031	397.2	16	0.203	0.047	3/119 (2.5%)
logoff	2470	46	1145	306	3.047	2.078	0.610	817.8	19	1.906	0.094	37/306 (12%)
setup	4751	67	1862	589	5.578	1.406	0.484	1917.8	64	1.157	0.063	34/589 (5.8%)

Fig. 1. WPDS experiments. The columns show the number of instructions (instrs); the number of procedures (CFGs); the number of basic blocks (BBs); the number of branch instructions (branches); the times, in seconds, for MOS and KS WPDS construction, running post*, and finding one-vocabulary affine relations at blocks that end with branch instructions, as well as the number of WPDS rules for which KS-weight generation timed out (t/o); and the degree of improvement gained by using $\widehat{\alpha}_{\text{KS}}$-generated KS weights rather than TSL-generated MOS weights (measured as the number of basic blocks that (i) end with a branch instruction, and (ii) begin with a node whose inferred one-vocabulary affine relation was strictly more precise under KS-based analysis).

We used EWPDS merge functions [15] to preserve caller-save and callee-save registers across call sites. The post* query used the FWPDS algorithm [14].

Fig. 1 lists several size parameters of the examples (number of instructions, procedures, basic blocks, and branches) along with the times for constructing abstract transformers and running post*.[2] Column 10 of Fig. 1 shows the number of WPDS rules for which KS-weight generation timed out. During WPDS construction, if Yices times out during $\widehat{\alpha}_{\text{KS}}$, the implementation creates the MOS weight for the rule instead, and then converts it to an over-approximating KS weight (§4.2). The number of rules equals the number of basic blocks plus the number of branches, so a timeout occurred for about 0.3–2.6% of the rules (geometric mean: 1.1%).

The experiment showed that the cost of constructing transformers via an SMT solver is high: creating the KS weights via $\widehat{\alpha}_{\text{KS}}$ is about 185 times slower than creating MOS weights using TSL (computed as the geometric mean of the construction-time ratios).

To address question 2, we performed two experiments:

– On a corpus of 11,144 instances of x86 instructions, we compared (i) the KS transformer created by applying $\widehat{\alpha}_{\text{KS}}$ to a quantifier-free bit-vector (QFBV) formula that captures the precise bit-level semantics of an instruction against

[2] Due to the high cost of the KS-based WPDS construction, we ran all analyses without the code for libraries. Values are returned from x86 procedure calls in register `eax`, and thus library functions were modeled approximately (albeit unsoundly, in general) by "`eax := ?`", where "`?`" denotes an unknown value [18] (sometimes written as "`havoc(eax)`").

(ii) the MOS transformer created for the instruction by the operator-by-operator reinterpretation method supported by TSL [16].

- We compared the precision improvement gained by using $\widehat{\alpha}_{\mathrm{KS}}$-generated KS weights rather than TSL-generated MOS weights in the WPDS-based analyses used to answer question 1. In particular, column 13 of Fig. 1 reports the number of basic blocks that (i) end with a branch instruction, and (ii) begin with a node whose inferred one-vocabulary affine relation was strictly more precise under KS-based analysis.

The first precision experiment showed that the $\widehat{\alpha}_{\mathrm{KS}}$ method is strictly more precise for about 8.3% of the instructions—910 out of the 11,022 instructions for which a comparison was possible. There were 122 Yices timeouts: 105 during $\widehat{\alpha}_{\mathrm{KS}}$ and 17 during the weight-comparison check.

Identical precision	KS more precise	Undetermined		Total
		Timeout during KS construction	Timeout during KS/MOS comparison	
10,112	910	105	17	11,144

Example 4. One instruction for which the $\widehat{\alpha}_{\mathrm{KS}}$-created transformer is better than the MOS transformer is "`add bh,al`", which adds the low-order byte of register `eax` to the second-to-lowest byte of register `ebx`. The transformer created by the TSL-based operator-by-operator reinterpretation method corresponds to `havoc`(`ebx`). All other registers are unchanged in both transformers—i.e., "$(\texttt{eax}' = \texttt{eax}) \wedge (\texttt{ecx}' = \texttt{ecx}) \wedge \ldots$". In contrast, the transformer obtained via $\widehat{\alpha}_{\mathrm{KS}}$ is

$$(2^{16}\texttt{ebx}' = 2^{16}\texttt{ebx} + 2^{24}\texttt{eax}) \wedge (\texttt{eax}' = \texttt{eax}) \wedge (\texttt{ecx}' = \texttt{ecx}) \wedge \ldots$$

Both transformers are over-approximations of the instruction's semantics, but the latter captures a relationship between the low-order two bytes of `ebx` and the low-order byte of `eax`, and hence is more precise.

The second precision experiment was based on the WPDS-based analyses used to answer question 1. The experiment showed that in our WPDS runs, the KS weights identified more precise one-vocabulary affine relations at about 12.3% of the basic blocks that end with a branch instruction (computed as the geometric mean of the precision ratios); see column 13 of Fig. 1.[3] In addition to the phenomenon illustrated in Ex. 4, two other factors contribute to the improved precision obtained via the KS domain:

- As discussed in §4.1 and §4.4, an MOS weight cannot express a transformation involving a guard on the pre-state vocabulary, whereas a KS weight can capture affine equality guards.

[3] Register `eip` is the x86 instruction pointer. There are some situations that cause the MOS weights to fail to capture the value of `eip` at a successor. Therefore, before comparing the affine relations computed via MOS and KS, we performed `havoc`(`eip`) so as to avoid biasing the results in favor of KS merely because of trivial affine relations of the form "$\texttt{eip} = \mathit{constant}$".

- To construct a KS weight w_{KS}^{B}, $\widehat{\alpha}_{\mathrm{KS}}$ is applied to $\varphi_{\mathcal{B}}$, which not only is bit-level precise, but also includes *all memory-access/update and flag-access/update operations*. Consequently, even though the KS weights we used in our experiments are designed to capture only transformations on registers, w_{KS}^{B} can account for transformations of register values that involve a sequence of memory and/or flag operations within a basic block.

The fact that $\varphi_{\mathcal{B}}$ can express dependences among registers that are mediated by one or more flag updates and flag accesses by instructions of a block, can allow a KS weight generated by $\widehat{\alpha}_{\mathrm{KS}}(\varphi_{\mathcal{B}})$ to sometimes capture `NULL`-pointer or return-code checks (as affine equality guards). For instance, the `test` instruction sets the zero flag `ZF` to *true* if the bitwise-and of its arguments is zero; the `jnz` instruction jumps if `ZF` is *false*. Thus,

- "`test esi, esi`; . . . `jnz xxx`" is an idiom for a `NULL`-pointer check: "`if(p == NULL)...`"
- "`call foo`; `test eax, eax`; . . . `jnz yyy`" is an idiom for checking whether the return value is zero: "`if(foo(...) == 0)...`".

The KS weights for the fall-through branches include the constraints "$\texttt{esi} = 0 \ \wedge\ \texttt{esi}' = 0$" and "$\texttt{eax} = 0 \ \wedge\ \texttt{eax}' = 0$", respectively, which both contain guards on the pre-state (i.e., "$\texttt{esi} = 0$" and "$\texttt{eax} = 0$", respectively). In contrast, the corresponding MOS weights—"$\texttt{esi}' = \texttt{esi}$" and "$\texttt{eax}' = \texttt{eax}$"—impose no constraints on the pre-state.

If a block $B = [I_1, \ldots, I_k]$ contains a spill to memory of register R_1 and a subsequent reload into R_2, the fact that w_{KS}^{B} is created from φ_B, which has a "global perspective" on the semantics of B, can—in principle—allow w_{KS}^{B} to capture the transformation $R_2' = R_1$. The corresponding MOS weight w_{MOS}^{B} would not capture $R_2' = R_1$ because the TSL-generated MOS weights are created by evaluating the semantic specifications of the individual instructions of B (over a domain of MOS values) and composing the results. Because each MOS weight $[\![I_j]\!]_{\mathrm{MOS}}$ in the composition sequence $w_{\mathrm{MOS}}^{B} := [\![I_k]\!]_{\mathrm{MOS}} \circ \ldots \circ [\![I_1]\!]_{\mathrm{MOS}}$ discards all information about how memory is transformed, the net effect on R_2' in w_{MOS}^{B} is $\texttt{havoc}(R_2')$. A second type of example involving a memory update followed by a memory access within a basic block is a sequence of the form "`push` *constant*; `pop esi`"; such sequences occur in several of the programs listed in Fig. 1.

Unfortunately, in our experiments Yices timed out on the formulas that arose in both kinds of examples, even with the timeout value set to 100 seconds.

7 Related Work

The original work on affine-relation analysis (ARA) was an intraprocedural ARA algorithm due to Karr [9]. Müller-Olm and Seidl introduced the MOS domain for affine relations, and gave an algorithm for interprocedural ARA [19,21]. King and Søndergaard defined the KS domain, and used it to create implementations of best abstract ARA transformers for the individual bits of a bit-blasted concrete semantics [11,12]. They used bit-blasting to express a bit-precise concrete semantics for a statement or basic block. The use of bit-blasting let them track the effect of non-linear bit-twiddling operations, such as shift and xor.

In this paper, we also work with a bit-precise concrete semantics; however, we avoid the need for bit-blasting by working with QFBV formulas expressed in terms of word-level operations; such formulas also capture the precise bit-level semantics of each instruction or basic block. We take advantage of the ability of an SMT solver to decide the satisfiability of such formulas, and use $\widehat{\alpha}_{\mathrm{KS}}$ to create best word-level transformers.

In contrast with both the Müller-Olm/Seidl and King/Søndergaard work, we take advantage of the Howell form of matrices. For each of the domains KS, AG, and MOS, because Howell form is canonical for non-empty sets of basis vectors, it provides a way to test pairs of elements for equality of their concretizations—an operation needed by analysis algorithms to determine when a fixed point is reached.

The algorithms given by Müller-Olm and Seidl avoid computing multiplicative inverses, which are needed to put a matrix in Howell form (line 8 of Alg. 1). However, their preference for algorithms that avoid inverses was originally motivated by the fact that at the time of their original 2005 work they were unaware [20] of Warren's $O(\log w)$ algorithms [26, §10-15] for computing the inverse of an odd element, and only knew of an $O(w)$ algorithm [19, Lemma 1].

Gulwani and Necula introduced the technique of random interpretation and applied it to identifying both intraprocedural [5] and interprocedural [6] affine relations. The fact that random interpretation involves collecting samples—which are similar to rows of AG elements—suggests that the AG domain might be used as an efficient abstract datatype for storing and manipulating data during random interpretation. Because the AG domain is equivalent to the KS domain (see §3), the KS domain would be an alternative abstract datatype for storing and manipulating data during random interpretation.

References

1. Bach, E.: Linear algebra modulo n. Unpublished manuscript (December 1992)
2. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. In: POPL (2003)
3. Dutertre, B., de Moura, L.: Yices: An SMT solver (2006), http://yices.csl.sri.com/
4. Elder, M., Lim, J., Sharma, T., Andersen, T., Reps, T.: Abstract domains of affine relations. Tech. Rep. TR-1691, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI (June 2011)
5. Gulwani, S., Necula, G.: Discovering affine equalities using random interpretation. In: POPL (2003)
6. Gulwani, S., Necula, G.: Precise interprocedural analysis using random interpretation. In: POPL (2005)
7. Hafner, J., McCurley, K.: Asymptotically fast triangularization of matrices over rings. SIAM J. Comput. 20(6) (1991)

8. Howell, J.: Spans in the module $(\mathbb{Z}_m)^s$. Linear and Multilinear Algebra 19 (1986)
9. Karr, M.: Affine relationship among variables of a program. Acta Inf. 6, 133–151 (1976)
10. Kidd, N., Lal, A., Reps, T.: WALi: The Weighted Automaton Library (2007), http://www.cs.wisc.edu/wpis/wpds/download.php
11. King, A., Søndergaard, H.: Inferring congruence equations using SAT. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 281–293. Springer, Heidelberg (2008)
12. King, A., Søndergaard, H.: Automatic abstraction for congruences. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 197–213. Springer, Heidelberg (2010)
13. Knoop, J., Steffen, B.: The interprocedural coincidence theorem. In: Pfahler, P., Kastens, U. (eds.) CC 1992. LNCS, vol. 641. Springer, Heidelberg (1992)
14. Lal, A., Reps, T.: Improving pushdown system model checking. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 343–357. Springer, Heidelberg (2006)
15. Lal, A., Reps, T., Balakrishnan, G.: Extended weighted pushdown systems. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 434–448. Springer, Heidelberg (2005)
16. Lim, J., Reps, T.: A system for generating static analyzers for machine instructions. In: Hendren, L. (ed.) CC 2008. LNCS, vol. 4959, pp. 36–52. Springer, Heidelberg (2008)
17. Meyer, C.: Matrix Analysis and Applied Linear Algebra. SIAM, Philadelphia (2000)
18. Müller-Olm, M., Seidl, H.: Precise interprocedural analysis through linear algebra. In: POPL (2004)
19. Müller-Olm, M., Seidl, H.: Analysis of modular arithmetic. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 46–60. Springer, Heidelberg (2005)
20. Müller-Olm, M., Seidl, H.: Personal communication (April 2005)
21. Müller-Olm, M., Seidl, H.: Analysis of modular arithmetic. TOPLAS 29(5) (2007)
22. Reps, T., Sagiv, M., Yorsh, G.: Symbolic implementation of the best transformer. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 252–266. Springer, Heidelberg (2004)
23. Reps, T., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. SCP 58(1-2) (October 2005)
24. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Program Flow Analysis: Theory and Applications. Prentice-Hall, Englewood Cliffs (1981)
25. Storjohann, A.: Algorithms for Matrix Canonical Forms. PhD thesis, ETH Zurich, Zurich, Switzerland, Diss. ETH No. 13922 (2000)
26. Warren Jr., H.: Hacker's Delight. Addison-Wesley, Reading (2003)

Transitive Closures of Affine Integer Tuple Relations and Their Overapproximations

Sven Verdoolaege, Albert Cohen, and Anna Beletska

INRIA and École Normale Supérieure

Abstract. The set of paths in a graph is an important concept with many applications in system analysis. In the context of integer tuple relations, which can be used to represent possibly infinite graphs, this set corresponds to the transitive closure of the relation representing the graph. Relations described using only affine constraints and projection are fairly efficient to use in practice and capture Presburger arithmetic. Unfortunately, the transitive closure of such a quasi-affine relation may not be quasi-affine and so there is a need for approximations. In particular, most applications in system analysis require overapproximations. Previous work has mostly focused either on underapproximations or special cases of affine relations. We present a novel algorithm for computing overapproximations of transitive closures for the general case of quasi-affine relations (convex or not). Experiments on non-trivial relations from real-world applications show our algorithm to be on average more accurate and faster than the best known alternatives.

1 Introduction

Computing the transitive closure of a relation is an operation underlying many important algorithms, with applications to computer-aided design, software engineering, scheduling, databases and optimizing compilers. In this paper, we consider the class of parametrized relations over integer tuples whose constraints consist of affine equalities and inequalities over variables, parameters and existentially quantified variables. This class has the same expressivity as Presburger arithmetic. Such quasi-affine relations typically describe infinite graphs, with the transitive closure corresponding to the set of all paths in the graph, and are widespread in decision and optimization problems with infinite domains, with applications to static analysis, formal verification and automatic parallelization [3, 4, 7, 15, 17, 18, 23, 25]. In this context, the use of quasi-affine relations is preferred because most operations on such relations can be performed exactly and fairly efficiently. However, as shown by Kelly et al. [25], the transitive closure of a quasi-affine relation may not be representable as a quasi-affine relation, or may not be computable at all. This leads to the design of approximation techniques [1, 6, 9, 24, 25]. and/or the study of sub-classes, including sub-polyhedral domains, where an exact computation is possible [2, 10–14, 18, 20]. Our approach belongs to the first group. That is, our goal is not to investigate classes of relations for which the transitive closure is guaranteed to be exact, but rather to obtain a general technique for quasi-affine relations that always produces an overapproximation, striking a balance between accuracy and speed.

E. Yahav (Ed.): SAS 2011, LNCS 6887, pp. 216–232, 2011.

```
for (i = 3; i <= n; i++)
  a[i] = f(a[i - 3]);
```

Fig. 1. A sequential loop

```
#pragma omp parallel for
for (i = 3; i <= min(n, 5); i++)
  for (j = i; j <= n; j += 3)
    a[j] = f(a[j - 3]);
```

Fig. 2. A parallelized version of the loop in Figure 1

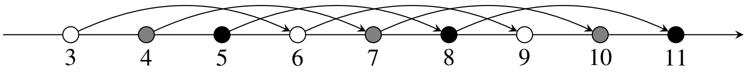


Fig. 3. The dependences and slices of the loop in Figure 1

Until recently, approximation for the general case of quasi-affine relations has only been investigated by Kelly et al. [25], and they focus on computing underapproximations.Yet the vast majority of the applications require overapproximations, and the algorithm proposed by Kelly et al. for computing overapproximations is very inaccurate, both in our own implementation and in an independent one [9]. Overapproximations have been considered by Beletska et al. [6], but in a more limited setting.

We use Iteration Space Slicing (ISS) [7] to illustrate the application of the transitive closure to quasi-affine relations. The objective of this technique is to split up the iterations of a loop nest into slices that can be executed in parallel. Applying the technique to the code in Figure 1, we see that some iterations of the loop use a result computed in earlier iterations and can therefore not be executed independently. These dependences are shown as arrows in Figure 3 for the case where $n = 11$. The intuition behind ISS is to group all iterations that are connected through dependences and to execute the resulting groups in parallel. These groups form a partition of the iterations, which is defined by an equivalence relation based on the dependences. In particular, the equivalence relation needs to be transitively closed, which requires the computation of the transitive closure of a relation representing the (extended) dependence graph. The resulting relation connects iterations to directly or indirectly depending iterations. In the example, three such groups can be discerned, indicated by different colors of the nodes in Figure 3. The resulting parallel program, with the outer parallel loop running over the different groups and the inner loop running over all iterations that belong to a group, is shown in Figure 2. It is important to note here that if the transitive closure cannot be computed exactly, then an overapproximation should be computed. This may result in more iterations being grouped together and therefore fewer slices and less parallelism, but the resulting program would still be correct. Underapproximation, on the other hand, would lead to invalid code. Furthermore, the overapproximation needs to be transitively closed since the slices should not overlap. Most of our target applications are based on dependence relations. For more information, we refer to our report [33].

In this paper, we present an algorithm for computing overapproximations of transitive closures. The algorithm subsumes those of [6] and [1]. Furthermore, it is experimentally shown to be exact in more instances from our applications than that of [25] and generally also faster on those instances where both produce an exact result. Our algorithm includes three decomposition methods, two of which are refinements of those of [25], while the remaining one is new. Finally, we provide a more extensive experimental evaluation on more difficult instances. As an indication, Kelly et al. [25] report that they were able to compute exact results for 99% of their input relations, whereas they can only compute exact results for about 60% of our input relations and our algorithm can compute exact results for about 80% of them. This difference in accuracy is shown to have an impact on the final outcome of some of our applications.

Section 2 gives background information on affine relations and transitive closures. We discuss related work in Section 3. Section 4 details the core of our algorithm. Section 5 studies a decomposition method to increase accuracy and speed. Section 6 describes the implementation. The results of our experiments are shown in Section 7.

2 Background

We use bold letters to denote vectors, and we write $\langle \mathbf{a}, \mathbf{x} \rangle$ for the scalar product of $\mathbf{a}$ and $\mathbf{x}$. We consider binary relations on $\mathbb{Z}^d$, i.e., relations mapping d-tuples of integers to d-tuples of integers. $S \circ R$ denotes the composition of two relations R and S. A relation R is *transitively closed* if $R \circ R = R$. The *transitive closure* of R, denoted R^+, is the (inclusion-wise) smallest transitively closed relation T such that $R \subseteq T$. The transitive closure R^+ can be constructed as the union of all positive integer powers of R:

$$R^+ := \bigcup_{k \geq 1} R^k, \quad \text{with} \quad R^k := \begin{cases} R & \text{if } k = 1 \\ R \circ R^{k-1} & \text{if } k \geq 2. \end{cases} \tag{1}$$

A relation R is *reflexively closed* on a set D if the identity relation Id_D is a subset of R. The *reflexive closure* of R on D is $R \cup \mathrm{Id}_D$. The *reflexive and transitive closure* of R on D is $R^*_D := R^+ \cup \mathrm{Id}_D$. The *cross product* of two relations R and S is the relation $R \times S = \{ (\mathbf{x}_1, \mathbf{y}_1) \to (\mathbf{x}_2, \mathbf{y}_2) \mid \mathbf{x}_1 \to \mathbf{x}_2 \in R \wedge \mathbf{y}_1 \to \mathbf{y}_2 \in S \}$. Occasionally, we will also consider binary relations over labeled integer tuples, i.e., subsets of $\bigcup_{d_1, d_2 \geq 0} (\Sigma \times \mathbb{Z}^{d_1}) \to (\Sigma \times \mathbb{Z}^{d_2})$, with Σ a finite set of labels. By assigning an integer value to each label, any such relation can be encoded as a relation over the $(1 + d)$-tuples with d the largest of the d_1s and d_2s over all elements in the relation.

We work with relations that have a finite representation. A commonly used class of such relations are those that can be represented using affine constraints. We consider finite unions of *basic relations* $R = \bigcup_i R_i$, each of which is represented as

$$R_i = \mathbf{s} \mapsto \{ \mathbf{x}_1 \to \mathbf{x}_2 \in \mathbb{Z}^d \times \mathbb{Z}^d \mid \exists \mathbf{z} \in \mathbb{Z}^e : A_1 \mathbf{x}_1 + A_2 \mathbf{x}_2 + B\mathbf{s} + D\mathbf{z} + \mathbf{c} \geq \mathbf{0} \}, \tag{2}$$

with $\mathbf{s}$ a vector of n free parameters, $A_i \in \mathbb{Z}^{m \times d}$, $B \in \mathbb{Z}^{m \times n}$, $D \in \mathbb{Z}^{m \times e}$ and $\mathbf{c} \in \mathbb{Z}^m$. The explicit declaration of the parameters ($\mathbf{s} \mapsto$) only serves to stress that the relation is parameteric and will sometimes be omitted. To emphasize that the description may involve existentially quantified variables $\mathbf{z}$, we call such relations *quasi-affine*. Any Presburger relation can be put in this form.

Unfortunately, the transitive closure of a quasi-affine relation may not be representable using affine constraints [25]. Similarly, a description of *all* positive integer powers k of R, parametrically in k, may not be representable either. We will refer to this description as simply the *power* of R and denote it as R^k. Since the power R^k as well as the transitive closure R^+ may not be representable, we will compute approximations, in particular overapproximations, denoted as $\mathcal{P}_k(R)$ and $\mathcal{T}(R)$, respectively.

Next to quasi-affine relations, our computations also make use of quasi-affine sets. These sets are defined essentially the same way: the only difference is that sets are unary relations on integer tuples instead of binary relations. The variables representing these tuples are called *set variables*. Sets can be obtained from relations in the following ways. The *domain* of a relation R is the set that results from *projecting out* the variables $\mathbf{x}_2$, i.e., $\operatorname{dom} R := \{ \mathbf{x}_1 \in \mathbb{Z}^d \mid \exists \mathbf{x}_2 \in \mathbb{Z}^d : \mathbf{x}_1 \to \mathbf{x}_2 \in R \}$. The *range* of a relation R is the set $\operatorname{ran} R := \{ \mathbf{x}_2 \in \mathbb{Z}^d \mid \exists \mathbf{x}_1 \in \mathbb{Z}^d : \mathbf{x}_1 \to \mathbf{x}_2 \in R \}$. The *difference set* of a relation R is the set $\Delta R := \{ \boldsymbol{\delta} \in \mathbb{Z}^d \mid \exists \mathbf{x} \to \mathbf{y} \in R : \boldsymbol{\delta} = \mathbf{y} - \mathbf{x} \}$. We also need the following operations on sets. The *Minkowski sum* of S_1 and S_2 is the set of sums of pairs of elements from S_1 and S_2, i.e., $S_1 + S_2 = \{ \mathbf{a} + \mathbf{b} \mid \mathbf{a} \in S_1 \wedge \mathbf{b} \in S_2 \}$. The kth multiple of a set, with k a positive integer is defined as $1 S = S$ and $k S = (k-1) S + S$ for $k \geq 2$. Note that as with the kth power of a relation, the kth multiple of a quasi-affine set, with k a parameter, may not be representable as a quasi-affine set.

Most of the applications we consider operate within the context of the polyhedral model [22], where single-entry single-exit program regions are represented, analyzed and transformed using quasi-affine sets and relations. In particular, the set of all iterations of a loop for which a given statement is executed is called the *iteration domain*. When the loop iterators are integers and lower and upper bounds of all enclosing loops as well all enclosing conditions are quasi-affine, then the iteration domain can be represented as a quasi-affine set. For example, the iteration domain of the single statement in Figure 1 is $n \mapsto \{ i \mid 3 \leq i \leq n \}$. *Dependence relations* map elements of an iteration domain to elements of another (or the same) iteration domain which depend on them for their execution. In the example, we have as dependence relation $n \mapsto \{ i \to i + 3 \mid 3 \leq i, i + 3 \leq n \}$. The graph with the statements and their iterations domains as nodes and the dependence relations as edges is called the *dependence graph*.

3 Related Work

The seminal work of Kelly et al. [25] introduced many of the concepts and algorithms in the computation of transitive closures that are also used in this paper. In particular, we use a revised version of their incremental computation and we apply their modified Floyd-Warshall algorithm internally. However, the authors consider a different set of applications which require underapproximations of the transitive closures instead of overapproximations. Their work therefore focuses almost exclusively on underapproximations. For overapproximations, they apparently consider some kind of "box-closure", which we recall in Section 6 and which is considerably less accurate than our algorithm.

Bielecki et al. [8] aim for exact results, which may therefore be non-affine. In our applications, affine results are preferred as they are easier to manipulate in further calculations. Furthermore, the authors only consider bijective relations over a convex domain. We consider general quasi-affine relations, which may be both non-bijective and defined over finite unions of domains.

Beletska et al. [6] consider finite unions of translations, for which they compute quasi-affine transitive closure approximations, as well as some other cases of finite unions of bijective relations, which lead to non-affine results. Their algorithm applied to unions of translations forms a special case of our algorithm for general affine relations.

Bielecki et al. [9] propose to compute the transitive closure using the classical iterative least fixed point computation and if this process does not produce the exact result after a fixed number of iterations, they resort to a variation of the "box-closure" of [25]. To increase the chances of the least fixed point computation, they first replace each disjunct in the input relation by its transitive closure, provided it can be computed exactly using available techniques [8, 25].

Transitive closures are also used in the analysis of counter systems to accelerate the computation of reachable sets. In this context, the power of a relation is known as a "counting acceleration"[20], while our relations over labeled tuples correspond to Presburger counter systems[20], extended to the integers. Much of the work on counter systems is devoted to the description of classes of systems for which the computations can be performed exactly. See, e.g., the work of Bardin et al.[2] and their references or the work of Bozga et al.[13]. By definition, these classes do not cover the class of input relations that we target in our approach. Other work on counter systems, e.g., that of Sankaranarayanan et al.[28], Feautrier and Gonnord[24] or Ancourt et al.[1], focuses on the computation of invariants and therefore allows for overapproximations. However, the analysis is usually performed on (non-parametric) polyhedra. That is, the relations for which transitive closures are computed do not involve parameters, existentially quantified variables or unions. The transitive closure algorithm proposed by Ancourt et al.[1] is essentially the same as that used by Boigelot and Herbreteau[12], except that the latter apply it on hybrid systems and only in cases where the algorithm produces an exact result. The same algorithm also forms the core of our transitive closure algorithm for single disjunct relations.

4 Powers and Transitive Closures

We present our core algorithm for computing overapproximations of the parametric power and the transitive closure of a relation. We first discuss the relationship between these two concepts and provide further evidence for the need for overapproximations. Then, we address the case where R is a single basic relation, followed by the case of multiple disjuncts. Finally, we explain how to check the exactness of the result and why the overapproximation is guaranteed to be transitively closed.

4.1 Introduction

There is a close relationship between parametric powers and transitive closures. Based on (1), the transitive closure R^+ can be computed from the parametric power R^k by projecting out the parameter k. Conversely, an algorithm for computing transitive closures can also be used to compute parametric powers. In particular, given a relation R, compute C^+ with $C = R \times \{\, i \to i + 1 \,\}$. For each pair of integer tuples in C, the difference between the final coordinates is 1. The difference between the final coordinates of pairs in C^+ is therefore equal to the number of steps taken. To compute R^k, one may equate k to this difference and subsequently project out the final coordinates.

As mentioned in Section 2, it is not always possible to compute powers and closures exactly, and we may aim instead for overapproximations $\mathcal{P}_k(R)$ and $\mathcal{T}(R)$. It should be clear that both conversions above map overapproximations to overapproximations. Important: note that the transitive closure may not be affinely representable even if the input relation is a union of constant-distance translations. A well know case can be built by considering the lengths of dependence paths associated to SUREs [19, Theorem 23].

4.2 Single Disjunct

Given a single basic relation R of the form (2), we look for an overapproximation of R^+ and we will derive it from an overapproximation of R^k. Furthermore, we want to compute the approximation efficiently and we want it to be as close to exact as possible.

We will treat the input relation as a (possibly infinite) union of translations. The distances covered by these translations are the elements of the difference set $\varDelta = \varDelta\, R$. We will assume here that $\varDelta$ also consists of a single basic set; our implementation of the $\varDelta\, R$ operation may result in a proper union due to our treatment of existentially quantified variables discussed below. The union case is treated in Section 4.3. Our approximation of the kth power contains translations over distances that are the sums of k distances in $\varDelta$. In particular, it contains those translations starting from and ending at the same points as those of the input relation. That is, we compute all paths along distances in $\varDelta$

$$P^k = \{\, \mathbf{x} \to \mathbf{y} \mid \exists \boldsymbol{\delta} \in \mathcal{D}^k : \mathbf{y} = \mathbf{x} + \boldsymbol{\delta} \,\}, \quad \text{with} \quad \mathcal{D}^k = k\varDelta \quad \text{and} \quad k \in \mathbb{Z}_{\geq 1}, \tag{3}$$

and intersect domain and range with those of R,

$$\mathcal{P}_k(R) = P^k \cap (\operatorname{dom} R \to \operatorname{ran} R)\,. \tag{4}$$

Example 1. To see the importance of this intersection with domain and range, consider the relation $R = \{\,(x, y) \to (x, x)\,\}$. First note that this relation is transitively closed already, so in our implementation we would not apply the algorithm here. If we did, however, then we would have $\varDelta\, R = \{\,0\,\} \times \mathbb{Z}$, whence $P^k = \{\,(x, y) \to (x, y')\,\}$. On the other hand, $\operatorname{ran} R = \{\,(x, x)\,\}$ and so $\mathcal{T}(R) = \mathcal{P}_k(R) = \{\,(x, y) \to (x, x)\,\}$.

Unfortunately, the set $k\varDelta$ in (3) may not be affine in general and then the same holds for P^k. As a trivial example of $k\varDelta$ not being affine, take $\varDelta$ to be the parametric singleton $n \to \{\,n\,\}$. If, however, $\varDelta$ is a non-parametric singleton $\varDelta = \{\,\boldsymbol{\delta}\,\}$, i.e., $\boldsymbol{\delta}$ does not depend on the parameters, then $k\varDelta$ is simply $\{\,k\boldsymbol{\delta}\,\}$ and we can compute our approximation of the power according to (4). Otherwise, we drop the definition of $\mathcal{D}^k$ in (3) and compute $\mathcal{D}^k$ as an approximation of $k\varDelta$, essentially copying some constraints of (a projection of) $\varDelta$. This process ensures that $\mathcal{D}^k$ is easy to compute, although it may in some cases not be the most accurate affine approximation of $k\varDelta$.

Let us first assume that the description of $\varDelta$ does not involve any existentially quantified variables or parameters. The constraints then have the form $\langle \mathbf{a}, \mathbf{x} \rangle + c \geq 0$. Any element in $k\varDelta$ can be written as the sum of k elements $\boldsymbol{\delta}_i$ from $\varDelta$. Each of these satisfies the constraint. The sum therefore satisfies the constraint

$$\langle \mathbf{a}, \mathbf{x} \rangle + c\,k \geq 0, \tag{5}$$

meaning that the constraint in (5) is valid for $k\Delta$. Our approximation $\mathcal{D}^k$ of $k\Delta$ is then the set bounded by the constraints in (5). In this special case, we compute essentially the same approximation as [1]. Note that if Δ has integer vertices, then the vertices of $\Delta \times \{ 1 \}$ generate the rational cone $\{ (\mathbf{x}, k) \in \mathbb{Q}^{d+1} \mid \langle \mathbf{a}, \mathbf{x} \rangle + c\,k \geq 0 \}$. This means that $\Delta \times \{ 1 \}$ is a Hilbert basis of this cone [29, Theorem 16.4] and that therefore $\mathcal{D}^k = k\Delta$.

Example 2. As a trivial example, consider the relation $R = \{ x \to y \mid 2 \leq y - x \leq 3 \}$. We have $\Delta = \Delta R = \{ \delta \mid 2 \leq \delta \leq 3 \}$ and $\mathcal{D}^k = k \mapsto \{ \delta \mid 2k \leq \delta \leq 3k \}$. Therefore, $\mathcal{P}_k(R) = P^k = k \mapsto \{ x \to y \mid 2k \leq y - x \leq 3k \}$ and $\mathcal{T}(R) = \{ x \to y \mid y - x \geq 2 \}$.

If the description of Δ does involve parameters, we cannot simply multiply the parametric constant by k: that would result in non-affine constraints. One option is to treat parameters as variables that just happen to remain constant. That is, instead of considering the set $\Delta = \Delta R = \mathbf{s} \mapsto \{ \boldsymbol{\delta} \in \mathbb{Z}^d \mid \exists \mathbf{x} \to \mathbf{y} \in R : \boldsymbol{\delta} = \mathbf{y} - \mathbf{x} \}$, we consider the set

$$\Delta' = \Delta R' = \{ \boldsymbol{\delta} \in \mathbb{Z}^{n+d} \mid \exists (\mathbf{s}, \mathbf{x}) \to (\mathbf{s}, \mathbf{y}) \in R' : \boldsymbol{\delta} = (\mathbf{s} - \mathbf{s}, \mathbf{y} - \mathbf{x}) \}. \tag{6}$$

The first n coordinates of every element in Δ' are zero. Projecting out these zero coordinates from Δ' is equivalent to projecting out the parameters in Δ. The result is obviously a superset of Δ, but all its constraints only involve the variables $\mathbf{x}$ and can therefore be treated as above.

Another option is to categorize the constraints of Δ according to whether they involve set variables, parameters or both. Constraints involving only set variables are treated as before. Constraints involving only parameters, i.e., constraints of the form

$$\langle \mathbf{b}, \mathbf{s} \rangle + c \geq 0. \tag{7}$$

are also valid for $k\Delta$. (Δ is empty for values of the parameters not satisfying these constraints and therefore so is $k\Delta$.) For constraints of the form

$$\langle \mathbf{a}, \mathbf{x} \rangle + \langle \mathbf{b}, \mathbf{s} \rangle + c \geq 0, \tag{8}$$

involving both set variables and parameters, we need to consider the sign of $\langle \mathbf{b}, \mathbf{s} \rangle + c$. If this expression is non-positive for all values of $\mathbf{s}$ for which Δ is non-empty, i.e.,

$$\Delta \cap \mathbf{s} \mapsto \{ \boldsymbol{\delta} \mid \langle \mathbf{b}, \mathbf{s} \rangle + c > 0 \} = \emptyset, \tag{9}$$

then $\langle \mathbf{a}, \mathbf{x} \rangle$ will always have a non-negative value v and we have $k \langle \mathbf{a}, \mathbf{x} \rangle \geq v$ for $k \geq 1$. The constraint in (8) is therefore also valid for $k\Delta$ if this condition holds. Our approximation $\mathcal{D}^k$ of $k\Delta$ is the set bounded by the constraints in (5), (7) and (8). Constraints of the form (8) for which (9) does not hold are simply dropped. Since this may result in a loss of accuracy, we add the constraints derived from Δ' above if any constraints of the form (8) get dropped.

Example 3. Consider the relation $R = n \to \{ (x, y) \to (1 + x, 1 - n + y) \mid n \geq 2 \}$. We have $\Delta R = n \to \{ (1, 1 - n) \mid n \geq 2 \}$ and so, by specifically treating parameters as described above, we obtain the following approximation for R^+: $n \to \{ (x, y) \to (x', y') \mid n \geq 2 \wedge y' \leq 1 - n + y \wedge x' \geq 1 + x \}$. If we consider instead $R' = \{ (n, x, y) \to$

$(n, 1 + x, 1 - n + y) \mid n \geq 2\,\}$ then $\Delta R' = \{\,(0, 1, y) \mid y \leq -1\,\}$ and we obtain the approximation $n \to \{\,(x, y) \to (x', y') \mid n \geq 2 \wedge x' \geq 1 + x \wedge y' \leq x + y - x'\,\}$. If we consider both ΔR and $\Delta R'$, then we obtain $n \to \{\,(x, y) \to (x', y') \mid n \geq 2 \wedge y' \leq 1 - n + y \wedge x' \geq 1 + x \wedge y' \leq x + y - x'\,\}$. Note however that this is not the most accurate affine approximation: $n \to \{\,(x, y) \to (x', y') \mid y' \leq 2 - n + x + y - x' \wedge n \geq 2 \wedge x' \geq 1 + x\,\}$ is a more accurate one.

If the description of Δ does involve existentially quantified variables, we compute unique representatives for these variables, picking the lexicographically minimal value for each of them using parametric integer programming [21]. The result is an explicit representation of each existentially quantified variable as the greatest integer part of an affine expression in the parameters and set variables. This representation may involve case distinctions, leading to a partitioning of Δ. If the representation involves only parameters, then the existentially quantified variable can be treated as a parameter. Similarly, if it only involves set variables, the existentially quantified variable can be treated as a set variable too. Otherwise, any constraints involving the variable are discarded. If this happens then, as before, we add the constraints derived from Δ' (6).

Example 4. Consider $R = n \to \{\,x \to y \mid \exists\, \alpha_0, \alpha_1 : 7\alpha_0 = -2 + n \wedge 5\alpha_1 = -1 - x + y \wedge y \geq 6 + x\,\}$. The difference set of this relation is $\Delta = \Delta R = n \to \{\,x \mid \exists\, \alpha_0, \alpha_1 : 7\alpha_0 = -2 + n \wedge 5\alpha_1 = -1 + x \wedge x \geq 6\,\}$. The existentially quantified variables can be defined in terms of the parameters and variables as $\alpha_0 = \lfloor(-2 + n)/7\rfloor$ and $\alpha_1 = \lfloor(-1 + x)/5\rfloor$. α_0 can therefore be treated as a parameter, while α_1 can be treated as a variable. This in turn means that $7\alpha_0 = -2 + n$ can be treated as a purely parametric constraint, while the other two constraints are non-parametric. The corresponding P^k is therefore $(n, k) \to \{\,x \to y \mid \exists\, \alpha_0, \alpha_1, f : k \geq 1 \wedge y = x + f \wedge 7\alpha_0 = -2 + n \wedge 5\alpha_1 = -k + f \wedge f \geq 6k\,\}$. Projecting out the parameter k and simplifying the result, we obtain the exact transitive closure $R^+ = n \to \{\,x \to y \mid \exists \beta_0, \beta_1 : 7\beta_0 = -2 + n \wedge 6\beta_1 \geq -x + y \wedge 5\beta_1 \leq -1 - x + y\,\}$.

4.3 Multiple Disjuncts

When the set of distances Δ is a proper union of basic sets $\Delta = \bigcup_i \Delta_i$, we apply the technique of Section 4.2 to each Δ_i separately, yielding approximations $\mathcal{D}_i^k$ of $k_i \Delta_i$ and corresponding paths P_i^k from (3). The set of global paths should take a total of k steps along the Δ_is, which can be obtained by essentially composing the P_i^ks and taking k to be the sum of all k_is. However, we need to allow for some k_is to be zero, so we introduce stationary paths $S_i = \mathrm{Id}_{\mathbb{Z}^d} \cap \{\,\mathbf{x} \to \mathbf{y} \mid k_i = 0\,\}$ and compute the set of global paths as

$$P^k = \Big((P_m^{k_m} \cup S_m) \circ \cdots \circ (P_2^{k_2} \cup S_2) \circ (P_1^{k_1} \cup S_1)\Big) \cap \{\,\mathbf{x} \to \mathbf{y} \mid k = \sum_i k_i > 0\,\}. \tag{10}$$

The final constraint ensures that at least one step is taken. The approximation of the power is then again computed according to (4). As explained in Section 4.1, $\mathcal{P}_k(R)$ can be represented as $\mathcal{T}(C)$, with $C = R \times \{\,i \to i + 1\,\}$. Using this representation, all Δ_i have 1 as their final coordinate and S_i above is simply $\mathrm{Id}_{\mathbb{Z}^{d+1}}$.

We need to be careful about scalability at this point. Given a set of distances Δ with m disjuncts, a naive application of (10) results in a P^k relation with $2^m - 1$ disjuncts. We try to limit this explosion in three ways. First, we handle all singleton Δ_i together; second, we try to avoid introducing a union with S_i; and third, we try to combine disjuncts. In particular, the paths along $\Delta_i = \{\boldsymbol{\delta}_i\}$ can be computed as

$$P^k = \{\mathbf{x} \to \mathbf{y} \mid \exists k_i \in \mathbb{Z}_{\geq 0} : \mathbf{y} = \mathbf{x} + \sum_i k_i \boldsymbol{\delta}_i \wedge \sum_i k_i = k > 0\}.$$

In this special case, we compute essentially the same approximation as [6]. For the remaining Δ_i, if the result of replacing constraint $k \geq 1$ by $k = 0$ in the computation of P^k yields the identity mapping, then $P_i^k \cup S_i$ is simply Q_i^k with Q_i^k the result of replacing $k \geq 1$ by $k \geq 0$. It is tempting to always replace $P_i^k \cup S_i$ by this Q_i^k, even if it is an overapproximation, but experience has shown that this leads to a significant loss in accuracy. Finally, if neither of these optimizations apply, then after each composition in (10) we "coalesce" the resulting relation. Coalescing detects pairs of disjuncts that can be replaced by a single disjunct without introducing any spurious elements [32].

4.4 Properties

By construction (Section 4.2 and Section 4.3), we have the following lemma.

Lemma 1. *$\mathcal{P}_k(R)$ is an overapproximation of R^k, i.e., $R^k \subseteq \mathcal{P}_k(R)$.*

The transitive closure approximation is obtained by projecting out the parameter k. If $\mathcal{P}_k(R)$ is represented as $\mathcal{T}(C)$, with $C = R \times \{i \to i + 1\}$, then $\mathcal{T}(R)$ can be obtained from $\mathcal{T}(C)$ by projecting out the final coordinates. The following lemma immediately holds.

Lemma 2. *$\mathcal{T}(R)$ is an overapproximation of R^+, i.e., $R^+ \subseteq \mathcal{T}(R)$.*

In many cases, $\mathcal{P}_k(R)$ will be exactly R^k. Given a particular R it is instructive to know whether the computed $\mathcal{P}_k(R)$ is exact or not, either for applications working directly with powers or as a basis for an exactness test on closures detailed below. The exactness test on powers amounts to checking whether $\mathcal{P}_k(R)$ satisfies the definition of R^k in (1):

$$\mathcal{P}_1(R) \subseteq R \quad \text{and} \quad \mathcal{P}_k(R) \subseteq R \circ \mathcal{P}_{k-1}(R) \text{ for } k \geq 2.$$

The reverse inclusion is guaranteed by Lemma 1. If $\mathcal{P}_k(R)$ is exact, then $\mathcal{T}(R)$ is also exact since the projection is performed exactly. However, if $\mathcal{P}_k(R)$ is *not* exact then $\mathcal{T}(R)$ might still be exact. We therefore prefer the more accurate test of [25, Theorem 5]:

$$\mathcal{T}(R) \subseteq R \cup (R \circ \mathcal{T}(R)).$$

However, this test can only be used if R is acyclic, i.e., if R^+ has no fixed points. Since $\mathcal{T}(R)$ is an overapproximation of R^+, it is sufficient to check that $\mathcal{T}(R)$ has no fixed points, i.e., that $\mathbf{0} \notin \Delta\,\mathcal{T}(R)$. If $\mathcal{T}(R)$ does have fixed points, then we apply the exactness test on $\mathcal{P}_k(R)$ instead.

Some applications also require the computed approximation of the transitive closure to be a transitively closed one [3, 7, 17]. The power approximation $\mathcal{P}_k(R)$ computed above is transitively closed as soon as P^k is transitively closed: if $\mathbf{x} \to \mathbf{y} \in \mathcal{P}_{k_1}(R)$

and $\mathbf{y} \to \mathbf{z} \in \mathcal{P}_{k_2}(R)$, then $\mathbf{x} \to \mathbf{z} \in \mathcal{P}_{k_1+k_2}(R)$, because P^k is transitively closed (and so $\mathbf{x} \to \mathbf{z} \in P^{k_1+k_2}$), $\mathbf{x} \in \operatorname{dom} R$ and $\mathbf{z} \in \operatorname{ran} R$. If $\mathbf{x}_1 \in \mathcal{D}^{k_1}$ and $\mathbf{x}_2 \in \mathcal{D}^{k_2}$, then both combinations satisfy (5) and so does their sum. Constraint (8) is also satisfied for $\mathbf{x}_1 + \mathbf{x}_2$, hence $\mathbf{x}_1 + \mathbf{x}_2 \in \mathcal{D}^{k_1+k_2}$. We conclude that in the single disjunct case, P^k in (3) is transitively closed, which in turn implies that also P^k in (10) is transitively closed in the multiple disjunct case. $\mathcal{T}(R)$ is transitively closed because for any $\mathbf{x} \to \mathbf{y}$ and $\mathbf{y} \to \mathbf{z}$ in $\mathcal{T}(R)$, there is some pair k_1, k_2 such that $\mathbf{x} \to \mathbf{y} \in \mathcal{P}_{k_1}(R)$ and $\mathbf{y} \to \mathbf{z} \in \mathcal{P}_{k_2}(R)$ and so $\mathbf{x} \to \mathbf{z} \in \mathcal{P}_{k_1+k_2}(R)$. We therefore have the following theorem.

Theorem 1. *$\mathcal{T}(R)$ is a transitively closed overapproximation of R^+.*

5 Strongly Connected Components

In order to improve accuracy, we apply several methods for breaking up the transitive closure computation. The first one is a decomposition into strongly connected components. The other two are variations of methods in [25]: we apply the modified Floyd-Warshall algorithm internally after partitioning the domain and we apply an incremental computation method. Our variations on these methods are explained in [33, Section 6].

Computations in Section 4.2 and Section 4.3 focus on the distance between elements in relation. The domain and range of the input relation are only taken into account at the very last step in (4). This means that translations described by one disjunct are applied to domain elements of other disjuncts, even if the domains are completely disjoint. In this section, we describe how the accuracy of $\mathcal{P}_k(R)$ and $\mathcal{T}(R)$ can be improved by decomposing the disjuncts of R into strongly connected components (SCCs).

The translations of R^+ are compositions of translations in the disjuncts of R. Two disjuncts R_i and R_j should be lumped into a connected component if there exist translations in R^k that first go through R_i and then through R_j, and translations that first go through R_j and then through R_i. Formally, we consider the directed graph whose vertices are the disjuncts in R and whose arcs connect pairs of vertices (R_i, R_j) if R_i can immediately follow R_j. The SCCs can be computed from this graph using Tarjan's algorithm [30]. In principle R_i can immediately follow R_j if the range of R_j intersects the domain of R_i, i.e., if $R_i \circ R_j \neq \emptyset$. However, if $R_i \circ R_j \subseteq R_j \circ R_i$ then one may always interchange R_i and R_j in any sequence leading to an element of R^+ where R_i immediately follows R_j. It is therefore sufficient to introduce an edge between R_i and R_j only if

$$R_i \circ R_j \nsubseteq R_j \circ R_i. \tag{11}$$

Once the components have been obtained, we compute $\mathcal{T}(R_c)$ on each component R_c separately. These $\mathcal{T}(R_c)$ can be combined into a global $\mathcal{T}(R)$ in the same way the paths are combined in (10). The combination must be performed according to a topological ordering of the components, obtained as a byproduct of Tarjan's algorithm. The decomposition preserves the validity of Lemma 1. The exactness check of Section 4.4 is performed on each component separately. If the approximation turns out to be inexact for any of the components, then the entire result is marked inexact and the exactness check is skipped on the remaining components.

To ensure closedness of $\mathcal{T}(R)$, we need to make a minor modification. If we are to perform the decomposition based solely on criterion $R_i \circ R_j \neq \emptyset$, then the same property will also hold for the components and, because of (4), for the powers of the components, implying that the final result is also transitively closed. If (11) is ever used, however, then transitive closedness of the result is not guaranteed unless all computations are performed exactly. We therefore explicitly check whether the result is transitively closed when the computation is not exact and when (11) has been used. If the check fails, we recompute the result without a decomposition into SCCs.

6 Implementation Details

The algorithms described in the previous sections have been implemented in the `isl` library [32]. For details about the algorithms, the design and the implementation of this integer set library, the reader is referred to the documentation and dedicated paper: `http://freshmeat.net/projects/isl`. The `isl` library supports both a parametric power ($\mathcal{P}_k(R)$) and a transitive closure ($\mathcal{T}(R)$) operation. Most of the implementation is shared between the two operations. The transitive closure operation first checks if the input happens to be transitively closed already and, if so, returns immediately. Both operations then check for strongly connected components.Within each component, either the modified Floyd-Warshall algorithm is applied or an incremental computation is attempted, depending on whether the domain and range can be partitioned. For practical reasons, incremental computation of powers has not been implemented. In the case of the power or in case no incremental computation can be performed, the basic single or multiple disjunct algorithm is applied. The exactness test is performed on the result of this basic algorithm. In the case of the transitive closure, the final coordinates encoding the path lengths are projected out on the same result. In the case of the power, the final coordinates are only projected out at the very end, after equating their difference to the exponent parameter. The `isl` library has direct support for unions of relations over pairs of labeled tuples. When the transitive closure of such a union is computed, we first apply the modified Floyd-Warshall algorithm on a partition based on the label and tuple size. Each recursive call is then handled as described above.

We also implemented a variation of the "box-closure" of Kelly et al. [25], which is a simplified version of the algorithm in Section 4.2. They overapproximate Δ by a rectangular box, possibly intersected with a rectangular lattice, with the box having fixed (i.e., non-parametric), but possibly infinite, lower and upper bounds. This overapproximation therefore has only non-parametric constraints and the corresponding $\mathcal{D}^k$ can be constructed using some very specific instances of (5). This algorithm clearly results in an overapproximation of R^k and therefore, after projection, of R^+. To improve accuracy, we also apply their incremental algorithm, but only in case the result is exact. The `ApproxClosure` operation which appeared in very recent versions of `Omega+` applies a similar algorithm. The main differences are that it does not perform an incremental computation and that it computes a box-closure on each disjunct individually.

7 Experiments

In all our experiments, we have used `isl` version isl-0.05.1-125-ga88daa9, `Omega+` version 2.1.6 [16], `Fast` version 2.1, `Aspic` version 3.2 and the latest version of

StInG[28].[1] Version 2.1.6 of Omega+ provides three transitive closure operations: the original TransitiveClosure (TC), which computes an underapproximation of the transitive closure; ApproxClosure (AC), which computes an overapproximation of the reflexive and transitive closure; and calculateTransitiveClosure (CTC), which appears to first try the least fixed point algorithm of [9] and then falls back on ApproxClosure. The execution times of the Omega+ transitive closure operations include the time taken for an extra exactness test. For TransitiveClosure, this test is based on [25, Theorem 1]. Presumably, a similar exactness test is performed internally, but the result of this test is not available to the user. In some cases, Omega+ returns a result containing UNKNOWN constraints and then it is clear that the result is not exact. In other cases, the user has no way of knowing whether the result is exact except by explicitly applying an exactness test. The isl library, by contrast, returns the exactness as an extra result. For ApproxClosure, we apply the test of [25, Theorem 5]. Note that this test may result in false positives when applied to cyclic relations. The exactness of the Aspic results is evaluated in the same way. Recall from Section 4.4 that we do not apply this test inside isl on relations that may be cyclic. Since it is not clear whether calculateTransitiveClosure will always produce an overapproximation, we apply both tests when checking its exactness. For the Fast results, no exactness test is needed since Fast will only terminate if it has computed an exact result. On the other hand, the execution time of Fast includes a conversion of the resulting Armoise formula to a quasi-affine relation, i.e., a disjunctive normal form. Since Fast only supports non-negative variables, we split all variables into a pair of non-negative variables whenever the input relation contains any negative value. Below, we discuss our experiments on inputs from our target domains. We have also performed some experiments [33] on the Aspic and Lever[31] test sets. On the first, isl performs comparably to Aspic, while on the second, isl only outperforms Lever on a small minority of the cases.

7.1 Type Size Inference

Chin and Khoo [17] apply the transitive closure operation to the following relation, derived from their Ackermann example: $\{\,(i,j) \to (i-1,j_1) \mid i \geq 1 \wedge j \geq 1\,\} \cup \{\,(i,j) \to (i,j-1) \mid i \geq 1 \wedge j \geq 1\,\} \cup \{\,(i,0) \to (i-1,1) \mid i \geq 1\,\}$. Omega produces an underapproximation and the authors heuristically manipulate this underapproximation to arrive at the following overapproximation: $\{\,(i,j) \to (i_1,j_1) \mid i_1 \geq 0 \wedge i_1 \leq i-1 \wedge j \geq 0\,\} \cup \{\,(i,j) \to (i,j_1) \mid j_1 \geq 0 \wedge j_1 \leq j-1 \wedge i \geq 1\,\}$. We compute the exact transitive closure: $\{\,(i,j) \to (o_0,o_1) \mid o_0 \geq 0 \wedge o_0 \leq -1+i \wedge j \geq 0 \wedge o_0 \leq -2+i+j\,\} \cup \{\,(i,j) \to (o_0,1) \mid o_0 \leq -1+i \wedge j \geq 0 \wedge o_0 \geq 0\,\} \cup \{\,(i,j) \to (i,o_1) \mid i \geq 1 \wedge o_1 \geq 0 \wedge o_1 \leq -1+j\,\} \cup \{\,(i,j) \to (o_0,0) \mid o_0 \leq -1+i \wedge j \geq 0 \wedge o_0 \geq 1\,\}$.

7.2 Equivalence Checking

Our most extensive set of experiments is based on the algorithm of [4] for checking the equivalence of a pair of static affine programs. Since the original implementation was not available to us, we have reimplemented the algorithm using VAUCANSON [26]

[1] The Fast and Aspic tests are based on the encoding described in [33, Section 8].

Table 1. Results for equivalence checking

			Omega+	
	isl	box	TC+AC	AC
proved equivalent	72	46	49	50
not proved equivalent	15	51	28	45
out-of-memory	17	12	14+18	4+12
time-out	9	4	4	2

Table 2. Outcome of transitive closure operations from equivalence checking

			Omega+					
	isl	box	TC	AC	CTC	Fast	Aspic	StInG
exact	472	334	366	267	274	139	201	215
inexact	67	227	157	266	245	0	268	240
failure	34	12	50	40	54	434	104	118

to compute regular expressions and isl to perform all set and relation manipulations. For the transitive closure operation we use the algorithm presented in this paper, the "box" implementation described in Section 6 or one of the implementations in Omega+. Since it is not clear whether calculateTransitiveClosure will always produce an overapproximation, we did not test this implementation in this experiment. The equivalence checking procedure requires overapproximations of transitive closures and using calculateTransitiveClosure might therefore render the procedure unsound. Since TransitiveClosure computes an underapproximation, we only use the results if they are exact. If not, we fall back on ApproxClosure. We will refer to this implementation as "TC+AC". For the other methods, we omit the exactness test in this experiment.

The equivalence checking procedure was applied to the output of CLooG [5] on 113 of its tests. In particular, the output generated when using the isl backend was compared against the output when using the PPL backend. These outputs should be equivalent for all cases, as was confirmed by the equivalence checking procedure of [34]. Table 1 shows the results. Using isl, 72 cases could be proven equivalent, while using Omega+ this number was reduced to only 49 or 50. This does not necessarily mean that all transitive closures were computed exactly; it just means that the results were accurate enough to prove equivalence. In fact, using ApproxClosure on its own, we can prove one more case equivalent than first using TransitiveClosure and then, if needed, ApproxClosure. On the other hand, as we will see below, TransitiveClosure is generally more accurate than ApproxClosure. A time limit of 1 hour was imposed, resulting in some cases timing out, and memory usage was capped at 2GB, similarly resulting in some out-of-memory conditions. For the Omega+ cases, we distinguish the real out-of-memory and maxing out the number of constraints (2048). The isl library does not impose a limit on the number of constraints. For those cases that Omega+'s ApproxClosure was able to handle (a strict subset of those that could be handled by isl), Figure 4 compares the running times. Surprisingly, isl is faster than Omega+'s ApproxClosure in all but one case. What is no surprise is that the running times (not shown in the figure) of the combined TransitiveClosure and ApproxClosure method are much higher still because it involves an explicit exactness test.

In order to compare the relative performance of the transitive closure operations themselves, we collected all transitive closure instances required in the above experiment. This resulted in a total of 573 distinct cases. The results are shown in Table 2, where failure may be out-of-memory (1GB), time-out (60s), or in case of Omega+, maxing out the number of constraints. Since only isl, box and Fast give an indication of whether the computed result is exact or not the results of the other methods are

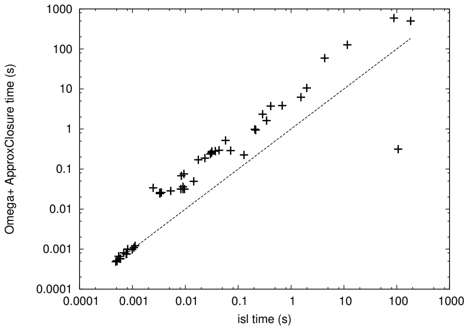


Fig. 4. Equivalence checking time

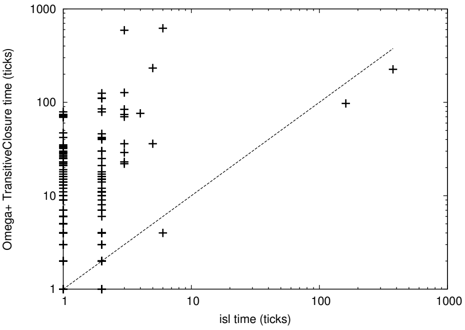


Fig. 5. Transitive closure computation time

explicitly checked for exactness. This exactness test may also contribute to some failures. Interestingly, our "box" implementation is more accurate than both `ApproxClosure` and `calculateTransitiveClosure` on this test set. On average, the `isl` implementation is more accurate than any of the `Omega+` implementations on the test set. There are also some exceptions, however. There are two cases where one or two of the `Omega+` implementations computes an exact result while both `isl` and the box implementation do not. In all those cases where `isl` fails, the other implementations either also fail or compute an inexact result. This observation, together with the higher failure rate (compared to the box implementation), suggests that our algorithm may be trying a little bit too hard to compute an exact result.

Figure 5 shows that for those transitive closures that both `TransitiveClosure` and `isl` compute exactly, `isl` is as fast as or faster than `Omega+` in all but a few exceptional cases. This result is somewhat unexpected since `Omega+`'s `TransitiveClosure` performs its operations in machine precision, while `isl` performs all its operations in exact integer arithmetic using GMP.

7.3 Iteration Space Slicing

The ISS experiments were performed on loop nests previously used in [7] and extracted from version 3.2 of NAS Parallel Benchmarks [35] consisting of five kernels and three

Table 3. Success rate of transitive closure operations from ISS experiment

		top-level								nested							
				Omega+								Omega+					
		isl	box	TC	AC	CTC	Fast	Aspic	StInG	isl	box	TC	AC	CTC	Fast	Aspic	StInG
mem	exact	70	44	58	43	53	25	15	39	37	25	35	7	31	1	1	15
	inexact	7	60	11	50	6	0	87	22	10	42	17	50	19	0	67	43
	failure	57	30	65	41	75	109	32	73	21	1	16	11	18	67	0	10
val	exact	72	44	57	43	57	28	37	39	53	35	47	23	37	7	8	28
	inexact	2	73	26	56	12	0	41	22	12	41	20	48	33	0	59	36
	failure	60	17	51	35	65	106	56	73	12	1	10	6	7	70	10	13

pseudo-applications derived from computational fluid dynamics applications. In total, 257 loops could be analyzed, but 123 have no dependences. For each of the remaining 134 loops, a dependence graph was computed using either value based dependence analysis or memory based dependence analysis [27]. Each of these dependence graphs was encoded as a single relation and passed to the transitive closure operation. The results are shown in Table 3. Since the input encodes an entire dependence graph, isl is expected to produce more accurate results than Omega+ as isl implements Floyd-Warshall internally. We therefore also show the results on all the nested transitive closure operations computed during the execution of Floyd-Warshall. It should be noted, though, that isl also performs coalescing on intermediate results, so an implementation of Floyd-Warshall on top of Omega+ may not produce results that are as accurate.

8 Conclusions and Future Work

We presented a novel algorithm for computing overapproximations of transitive closures for the general case of affine relations. The overapproximations computed by the algorithm are guaranteed to be transitively closed. The algorithm was experimentally shown to be significantly more accurate than the best known alternatives on representative benchmarks from our target applications, and our implementation is generally also faster despite performing all computations in exact integer arithmetic.

Although our algorithm can be applied to any affine relation, we have observed that the results are not very accurate if the input relation is cyclic. As part of future work, we therefore want to devise improved strategies for handling such cyclic relations. The comparison with tools for reachability or invariant analysis has revealed that our problems have quite different characteristics, in that our algorithm does not work very well on their problems while their algorithms do not work very well on ours. The design of a combined approach that could work for both classes of problems is therefore also an interesting line of research.

Acknowledgments. We would like to thank Louis-Noël Pouchet for showing us how to use VAUCANSON, Laure Gonnord for extending Aspic to produce isl compatible output and Jérôme Leroux for explaining how to encode relations in Fast.

References

1. Ancourt, C., Coelho, F., Irigoin, F.: A modular static analysis approach to affine loop invariants detection. Electron. Notes Theor. Comput. Sci. 267, 3–16 (2010)
2. Bardin, S., Finkel, A., Leroux, J., Petrucci, L.: Fast: acceleration from theory to practice. STTT 10(5), 401–424 (2008)
3. Barthou, D., Cohen, A., Collard, J.-F.: Maximal static expansion. Int. J. Parallel Programming 28(3), 213–243 (2000)
4. Barthou, D., Feautrier, P., Redon, X.: On the equivalence of two systems of affine recurrence equations. In: Monien, B., Feldmann, R.L. (eds.) Euro-Par 2002. LNCS, vol. 2400, pp. 309–313. Springer, Heidelberg (2002)
5. Bastoul, C.: Code generation in the polyhedral model is easier than you think. In: PACT 2004: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, pp. 7–16. IEEE Computer Society, Washington, DC, USA (2004)

6. Beletska, A., Barthou, D., Bielecki, W., Cohen, A.: Computing the transitive closure of a union of affine integer tuple relations. In: Du, D.-Z., Hu, X., Pardalos, P.M. (eds.) COCOA 2009. LNCS, vol. 5573, pp. 98–109. Springer, Heidelberg (2009)
7. Beletska, A., Bielecki, W., Cohen, A., Palkowski, M., Siedlecki, K.: Coarse-grained loop parallelization: Iteration space slicing vs affine transformations. In: International Symposium on Parallel and Distributed Computing, 73–80 (2009)
8. Bielecki, W., Klimek, T., Trifunovic, K.: Calculating exact transitive closure for a normalized affine integer tuple relation. Electronic Notes in Discrete Mathematics 33, 7–14 (2009)
9. Bielecki, W., Klimek, T., Palkowski, M., Beletska, A.: An iterative algorithm of computing the transitive closure of a union of parameterized affine integer tuple relations. In: Wu, W., Daescu, O. (eds.) COCOA 2010, Part I. LNCS, vol. 6508, pp. 104–113. Springer, Heidelberg (2010)
10. Boigelot, B.: Symbolic Methods for Exploring Infinite State Spaces. Ph.D. thesis, Université de Liège (1998)
11. Boigelot, B., Wolper, P.: Symbolic verification with periodic sets. In: Proceedings of the 6th International Conference on Computer-Aided Verification. Lecture Notes in Computer Science, vol. 818, pp. 55–67. Springer, Heidelberg (1994)
12. Boigelot, B., Herbreteau, F.: The power of hybrid acceleration. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 438–451. Springer, Heidelberg (2006)
13. Bozga, M., Gîrlea, C., Iosif, R.: Iterating octagons. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 337–351. Springer, Heidelberg (2009)
14. Bozga, M., Iosif, R., Konečný, F.: Fast acceleration of ultimately periodic relations. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 227–242. Springer, Heidelberg (2010)
15. Bultan, T., Gerber, R., Pugh, W.: Model-checking concurrent systems with unbounded integer variables: symbolic representations, approximations, and experimental results. ACM Trans. Program. Lang. Syst. 21(4), 747–789 (1999)
16. Chen, C.: Omega+ library (2009), http://www.chunchen.info/omega/
17. Chin, W.N., Khoo, S.C.: Calculating sized types. Higher Order Symbol. Comput. 14(2-3), 261–300 (2001)
18. Comon, H., Jurski, Y.: Multiple counters automata, safety analysis and presburger arithmetic. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 268–279. Springer, Heidelberg (1998)
19. Darte, A., Robert, Y., Vivien, F.: Scheduling and Automatic Parallelization. Birkhauser, Boston (2000)
20. Demri, S., Finkel, A., Goranko, V., van Drimmelen, G.: Towards a model-checker for counter systems. In: Graf, S., Zhang, W. (eds.) ATVA 2006. LNCS, vol. 4218, pp. 493–507. Springer, Heidelberg (2006)
21. Feautrier, P.: Parametric integer programming. Operationnelle/Operations Research 22(3), 243–268 (1988)
22. Feautrier, P.: Automatic Parallelization in the Polytope Model. In: Perrin, G.-R., Darte, A. (eds.) The Data Parallel Programming Model. LNCS, vol. 1132, pp. 79–100. Springer, Heidelberg (1996)
23. Feautrier, P., Griebl, M., Lengauer, C.: On index set splitting. In: Parallel Architectures and Compilation Techniques, PACT 1999, Newport Beach, CA (October 1999)
24. Feautrier, P., Gonnord, L.: Accelerated invariant generation for c programs with aspic and c2fsm. Electron. Notes Theor. Comput. Sci. 267, 3–13 (2010)
25. Kelly, W., Pugh, W., Rosser, E., Shpeisman, T.: Transitive closure of infinite graphs and its applications. In: Huang, C.H., Sadayappan, P., Banerjee, U., Gelernter, D., Nicolau, A., Padua, D.A. (eds.) LCPC 1995. LNCS, vol. 1033, pp. 126–140. Springer, Heidelberg (1996)
26. Lombardy, S., Régis-Gianas, Y., Sakarovitch, J.: Introducing VAUCANSON. Theor. Comput. Sci. 328(1-2), 77–96 (2004)

27. Pugh, W., Wonnacott, D.: An exact method for analysis of value-based array data dependences. In: Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing, pp. 546–566. Springer, Heidelberg (1994)
28. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Constraint-based linear-relations analysis. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 53–68. Springer, Heidelberg (2004)
29. Schrijver, A.: Theory of Linear and Integer Programming. John Wiley & Sons, Chichester (1986)
30. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM Journal on Computing 1(2), 146–160 (1972)
31. Vardhan, A., Viswanathan, M.: LEVER: A tool for learning based verification. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 471–474. Springer, Heidelberg (2006)
32. Verdoolaege, S.: *isl*: An integer set library for the polyhedral model. In: Fukuda, K., van der Hoeven, J., Joswig, M., Takayama, N. (eds.) ICMS 2010. LNCS, vol. 6327, pp. 299–302. Springer, Heidelberg (2010)
33. Verdoolaege, S., Cohen, A., Beletska, A.: Transitive closures of affine integer tuple relations and their overapproximations. Tech. Rep. RR-7560, INRIA (March 2011)
34. Verdoolaege, S., Janssens, G., Bruynooghe, M.: Equivalence checking of static affine programs using widening to handle recurrences. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 599–613. Springer, Heidelberg (2009)
35. NAS benchmarks suite, `http://www.nas.nasa.gov`

Logico-Numerical Abstract Acceleration and Application to the Verification of Data-Flow Programs[⋆]

Peter Schrammel and Bertrand Jeannet

INRIA Rhône-Alpes, Grenoble, France
{peter.schrammel,bertrand.jeannet}@inria.fr

Abstract. Acceleration methods are commonly used for speeding up the convergence of loops in reachability analysis of counter machine models. Applying these methods to synchronous data-flow programs with Boolean and numerical variables, *e.g.*, Lustre programs, requires the enumeration of the Boolean states in order to obtain a control flow graph (CFG) with numerical variables only. Our goal is to apply acceleration techniques to data-flow programs without resorting to this exhaustive enumeration. To this end, we present (1) *logico-numerical abstract acceleration methods* for CFGs with Boolean and numerical variables and (2) partitioning techniques that make logical-numerical abstract acceleration effective. Experimental results show that incorporating these methods in a verification tool based on abstract interpretation provides not only significant advantage in terms of accuracy, but also a gain in performance in comparison to standard techniques.

Keywords: Verification, Static Analysis, Abstract Interpretation, Abstract Acceleration, Control Flow Graph Partitioning.

1 Introduction

This paper deals with the *verification of safety properties* about *logico-numerical* data-flow programs, *i.e.*, programs manipulating Boolean and numerical variables. Verification of such properties amounts to checking whether the reachable state space stays within the invariant specified by the property.

Classical applications are safety-critical controllers as found in modern transport systems, as well as static checking of high-level simulation models, *e.g.* a model of a production line as depicted in Fig. 1. In such systems the properties to be proved, like throughput and workload, depend essentially on the relationships between the numerical variables of the system. Yet, there is an important observation that we are going to exploit: In many of these control systems large parts of the program simply count time or events, or, more generally, they perform rather regular linear arithmetic operations. Hence, it is appropriate to take advantage of a specialized analysis method that exploits this regularity in order

[⋆] This work was supported by the INRIA large-scale initiative Synchronics.

E. Yahav (Ed.): SAS 2011, LNCS 6887, pp. 233–248, 2011.

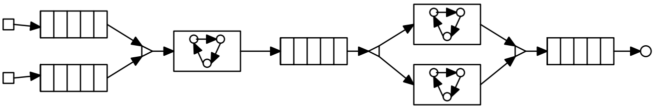


Fig. 1. Example of a production line with buffers, machines, and splitting and combining material flows

to improve verification performance and precision. In this paper, we will consider abstract acceleration [1] for this purpose, which aims at computing in one step the effect of an unbounded number of loop iterations. However, at the same time, we are confronted with a huge Boolean state space in the applications we want to verify. Our contribution is therefore to extend abstract acceleration from purely numerical programs to logico-numerical programs in an efficient way.

Verifying logico-numerical data-flow programs by abstract interpretation. The reachability problem is not decidable for this class of programs, so analysis methods are incomplete. Abstract interpretation [2] is a classical method with guaranteed termination for the price of an approximate analysis result. The key idea is to approximate sets of states S by elements $S^\sharp$ of an *abstract domain.* A classical abstract domain for numerical invariants in $\wp(\mathbb{R}^n)$ is the domain of convex polyhedra $Pol(\mathbb{R}^n)$ [3]. An approximation $S^\sharp$ of the reachable set S is then computed by iteratively solving the fixed point equation characterizing S *in the abstract domain.* To ensure termination when the abstract domain contains infinitely increasing chains, one applies an extrapolation operator called *widening*, which induces additional approximations.

Since the analysis with a single abstract value gives only coarse results, it is usually conducted over a *control flow graph* (CFG) of the program. In the case of imperative programs, such a control graph can be obtained easily by associating control points with programming constructs as if-then-else or while. Data-flow programs do not have such constructs; yet, one can use finite-type variables such as Booleans to generate a control structure. Thus, the classical approach is to explicitly unfold the Boolean control structure by *enumerating* the Boolean state space and to analyze the numerical variables on the obtained CFG using a numerical abstract domain. The problem is that the analysis becomes intractable with larger programs because the number of control locations grows exponentially with the number of Boolean states.

Jeannet [4] proposed a method for iteratively refining the control structure and analyzing the system using a *logico-numerical abstract domain*, making it possible to deal with Boolean variables symbolically. We want to complement this approach with new partitioning techniques and analysis methods.

Abstract acceleration. *Acceleration* [5] refers to a set of techniques aiming at exactly computing the effects of loops in numerical transition systems like counter machines, and ultimately at computing the exact reachability set of such systems, usually using Presburger arithmetic. *Abstract acceleration* [1] reformulates these concepts within an abstract interpretation approach: it aims at computing the best correct approximation of the effect of loops in a given abstract domain (currently only convex polyhedra have been considered).

These techniques can analyze only purely numerical programs with a given CFG, of which the size often becomes prohibitively large. Furthermore, they do not consider numerical inputs. In a previous paper [6], we already extended abstract acceleration to numerical inputs.

Contributions. The missing link in the application of abstract acceleration to logico-numerical programs, such as Lustre programs, is an efficient method for (i) building an appropriate CFG without resorting to Boolean state space enumeration, and (ii) analyzing it using abstract acceleration. Our methods allow us to treat these two problems independently of each other.

Our contributions can be summarized as follows:

1. We propose methods for *accelerating self-loops* in the CFG of *logico-numerical* data-flow programs.
2. We define *Boolean partitioning heuristics* that favor the applicability of abstract acceleration and enable a reasonably precise reachability analysis.
3. We provide *experimental results* on the use of abstract acceleration enhancing the analysis of logico-numerical programs.

Compared to other approaches, the partitioning heuristics that we propose are based on structural properties of the program, namely the numerical transitions, and thus, they are complementary to most common techniques based on abstract or concrete counter-example refinement. In this paper we consider only partitions of the Boolean state space, in contrast to the tool NBac [4], which in addition partitions according to numerical constraints.

Organisation of the article. §2 gives an introduction to the abstract interpretation of logico-numerical programs, partitioning, and abstract acceleration. §3 and §4 describe our contributions on logico-numerical abtract acceleration methods, §5 presents our experimental results, and finally §6 discusses related work and concludes.

2 Analysis of Logico-Numerical Programs

Program model. We consider programs modeled as a symbolic transition system $\begin{cases}\mathcal{I}(\boldsymbol{s})\\ \mathcal{A}(\boldsymbol{s},\boldsymbol{i}) \to \boldsymbol{s}' = \boldsymbol{f}(\boldsymbol{s},\boldsymbol{i})\end{cases}$ where (1) $\boldsymbol{s}$ and $\boldsymbol{i}$ are vectors of state and input variables, that are either Boolean or numerical; (2) $\mathcal{I}(\boldsymbol{s})$ is an initial condition on state variables; (3) $\mathcal{A}(\boldsymbol{s},\boldsymbol{i})$ is an *assertion* constraining input variables depending on state variables, and typically modeling the environment of the program; (4) $\boldsymbol{f}$ is the vector of transition functions. An example of such a program is

$$\begin{cases} \mathcal{I}(b,x) = \neg b \wedge (x=0) \\ 1 \le \xi \le 3 \to \begin{pmatrix} b' \\ x' \end{pmatrix} = \begin{pmatrix} (b \wedge x \le 5) \vee \beta \\ \begin{cases} x+\xi & \text{if } b \wedge x \le 5 \\ 0 & \text{otherwise} \end{cases} \end{pmatrix} \end{cases}$$

An execution of such a system is a sequence $\boldsymbol{s^0} \xrightarrow{\boldsymbol{i^0}} \boldsymbol{s^1} \xrightarrow{\boldsymbol{i^1}} \ldots \boldsymbol{s^k} \xrightarrow{\boldsymbol{i^k}} \ldots$ such that $\mathcal{I}(\boldsymbol{s^0})$ and for any $k \geq 0$, $\mathcal{A}(\boldsymbol{s^k}, \boldsymbol{i^k}) \wedge \boldsymbol{s^{k+1}} = \boldsymbol{f}(\boldsymbol{s^k}, \boldsymbol{i^k}))$.

The front-end compilation of synchronous data-flow programs, like Lustre, produces such a program model, that also includes various models of counter automata (by emulating locations using Boolean variables) [5].

We will use the following notations:

$\boldsymbol{s} = (\boldsymbol{b}, \boldsymbol{x})$: state variable vector, with $\boldsymbol{b}$ Boolean and $\boldsymbol{x}$ numerical subvectors
$\boldsymbol{i} = (\boldsymbol{\beta}, \boldsymbol{\xi})$: input variable vector, with $\boldsymbol{\beta}$ Boolean and $\boldsymbol{\xi}$ numerical subvectors
$\mathcal{C}(\boldsymbol{x}, \boldsymbol{\xi})$: constraints over numerical variables, seen as a vector of Boolean decisions (for short $\mathcal{C}$)

Transitions are written in the form $\mathcal{A}(\boldsymbol{b}, \boldsymbol{\beta}, \mathcal{C}) \rightarrow \begin{pmatrix} \boldsymbol{b}' \\ \boldsymbol{x}' \end{pmatrix} = \begin{pmatrix} \boldsymbol{f}^b(\boldsymbol{b}, \boldsymbol{\beta}, \mathcal{C}) \\ \boldsymbol{f}^x(\boldsymbol{b}, \boldsymbol{\beta}, \mathcal{C}, \boldsymbol{x}, \boldsymbol{\xi}) \end{pmatrix}$. Numerical transition functions are written as a disjunction of guarded actions: $\boldsymbol{f}^x(\boldsymbol{b}, \boldsymbol{\beta}, \mathcal{C}, \boldsymbol{x}, \boldsymbol{\xi}) = \bigvee_i \big(g_i(\boldsymbol{b}, \boldsymbol{\beta}, \mathcal{C}) \rightarrow \boldsymbol{a}_i^x(\boldsymbol{x}, \boldsymbol{\xi})\big)$ with $\neg(g_i \wedge g_j)$ for $i \neq j$. The program example above conforms to these notations.

2.1 Abstract Interpretation

The state space induced by logico-numerical programs has the structure $E = \mathbb{B}^m \times \mathbb{R}^n$. As mentioned in the introduction, we adopt the abstract interpretation framework so as to abstract the equation $S = S^0 \cup \mathit{post}(S), S \in \wp(E)$ in an abstract domain and to solve it iteratively, using widening to ensure convergence.

We consider the domain $A = \wp(\mathbb{B}^m) \times \mathit{Pol}(\mathbb{R}^n)$ of *convex states* [7], which approximates a set of states coarsely by a conjunction of a Boolean formula and a single convex polyhedron. For instance the formula $(b \wedge x \leq 2) \vee (\neg b \wedge x \leq 4)$ is abstracted by $\mathit{true} \wedge x \leq 4$.

Partitioning the state space. We use state space partitioning to obtain a CFG in which each equivalence class of the partition corresponds to a location.

Definition 1. *A symbolic control flow graph (CFG) of a symbolic transition system is a directed graph* $\langle \Pi, \Pi_0, \rightsquigarrow \rangle$ *where*

- Π *is the set of locations; each location* $\ell \in \Pi$ *is characterized by its location invariant* $\varphi_\ell(\boldsymbol{s})$*, such that* $\{\varphi_\ell(\boldsymbol{s}) \mid \ell \in \Pi\}$ *forms a partition of* E.
- Π_0 *is the set of initial locations with* $\mathcal{I}(\boldsymbol{s}) = \bigvee_{\ell \in \Pi_0} \varphi_\ell(\boldsymbol{s})$
- $\rightsquigarrow$ *defines arcs between locations according to the transition relation:* $\exists \boldsymbol{s}, \boldsymbol{i} : \varphi_\ell(\boldsymbol{s}) \wedge \mathcal{A}(\boldsymbol{s}, \boldsymbol{i}) \wedge \boldsymbol{s}' = \boldsymbol{f}(\boldsymbol{s}, \boldsymbol{i}) \wedge \varphi_{\ell'}(\boldsymbol{s}') \Rightarrow \ell \rightsquigarrow \ell'$

There are several ways to define a partition inducing such a CFG. In predicate abstraction for instance, the partition is generated by considering the truth value of a finite set of predicates [8]. Here, we consider partitions defined by equivalence relations on Boolean state variables. For example, the fully partitioned CFG obtained by *enumerating* all Boolean states is characterized by the relation $\boldsymbol{b}_1 \sim \boldsymbol{b}_2 \Leftrightarrow \boldsymbol{b}_1 = \boldsymbol{b}_2$.

Simplifying a CFG. In practice, partitioning is done by incrementally dividing the locations. Furthermore arcs between locations that are proved to be

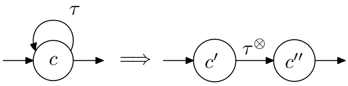


Fig. 2. Self-loop transition (left) and accelerated transition (right)

infeasible are removed. This can be done, *e.g.* by checking the satisfiability of the transition relation, *e.g.* using an SMT solver.

At last, transition functions are simplified by *partial evaluation* (using a generalized cofactor operator, see [9]).

Analyzing a CFG. In the context of analysis by abstract interpretation, considering a CFG allows to apply widening in a more restrictive way, *e.g.* on loop heads only [10]. Also the information loss due to the convex union is limited, because we assign an abstract value to each location: We consider the compound abstract domain $(\Pi \to A)$ where the concrete states S are connected to their abstract counterparts $S^\sharp$ by the *Galois connection*:

$$S^\sharp = \alpha(S) = \lambda\ell \,.\, \alpha(S \sqcap \varphi_\ell) \qquad\qquad S = \gamma(S^\sharp) = \bigcup_{\ell\in\Pi} \gamma(S^\sharp_\ell)$$

Analyzing the partitioned system amounts to computing the least fixed point $S^\sharp = S^{\sharp,0} \sqcup \lambda\ell \,.\, \bigsqcup_{\ell'\in\Pi} \Big(post(S^\sharp_{\ell'}) \sqcap \varphi_\ell\Big)$ where $S^\sharp, S^{\sharp,0} \in (\Pi \to A)$.

2.2 Abstract Acceleration

As mentioned in the introduction, *acceleration* [5] aims at computing exactly (or precisely in the case of abstract acceleration [1,11]) the effect of a self-loop. The basic idea is to replace a loop transition by its transitive closure (Fig. 2) by providing a formula $\tau^{\otimes}(X)$ computing $\tau^*(X) = \bigcup_{k\geq 0} \tau^k(X)$.

Basic concepts. A loop transition τ has the structure: $g \to a$ meaning "while guard g do action a". Our extension of abstract acceleration to numerical inputs [6] deals with loop transitions of the form

$$\underbrace{\begin{pmatrix}\mathbf{A} & \mathbf{L}\\ 0 & \mathbf{J}\end{pmatrix}\begin{pmatrix}\boldsymbol{x}\\ \boldsymbol{\xi}\end{pmatrix} \leq \begin{pmatrix}\boldsymbol{v}\\ \boldsymbol{k}\end{pmatrix}}_{\mathbf{A}\boldsymbol{x}+\mathbf{L}\boldsymbol{\xi}\leq\boldsymbol{v}\ \wedge\ \mathbf{J}\boldsymbol{\xi}\leq\boldsymbol{k}} \to \boldsymbol{x}' = \underbrace{\begin{pmatrix}\mathbf{C} & \mathbf{T}\end{pmatrix}\begin{pmatrix}\boldsymbol{x}\\ \boldsymbol{\xi}\end{pmatrix} + \boldsymbol{u}}_{\mathbf{C}\boldsymbol{x}+\mathbf{T}\boldsymbol{\xi}+\boldsymbol{u}} \tag{1}$$

Existing acceleration methods can deal with transitions where the matrix $\mathbf{C}$ is a diagonal matrix with zeros and ones only or when it is periodic ($\exists p > 0, l > 0 : \mathbf{C}^{p+l} = \mathbf{C}^p$). Throughout this paper, we will call such numerical transition functions *accelerable*, whereas we regard general affine transformations (with an arbitrary $\mathbf{C}$) as *non-accelerable.*

Widening and acceleration. Acceleration gives us a formula for computing the transitive closure of accelerable loop transitions. Widening is still needed in the case of non-accelerable transitions, outer loops of nested loops and to guarantee convergence when there are multiple self-loops in the same control location (see the concept of *flat systems* in [5]). The main advantages of abstract acceleration *in comparison with widening* result from two properties:

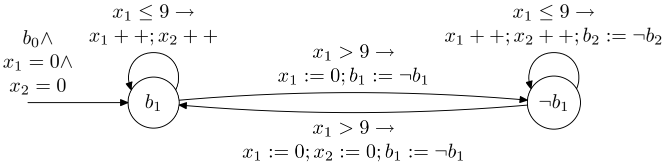


Fig. 3. Self-loop ready to be accelerated (left). Acceleration not applicable (right)

- *Idempotency* ($\tau^{\otimes}(X) = \tau^{\otimes}(\tau^{\otimes}(X))$), which simplifies the fixed point computation (widening usually requires more than one step to stabilize);
- *Monotonicity* $X_1 \sqsubseteq X_2 \Rightarrow \tau^{\otimes}(X_1) \sqsubseteq \tau^{\otimes}(X_2)$, that makes the analysis more robust and predictible (whereas widening operators are not monotonic).

2.3 Classical Application of Abstract Acceleration

We describe now the classical way to apply abstract acceleration to the analysis of logico-numerical programs, for which this paper proposes major enhancements.

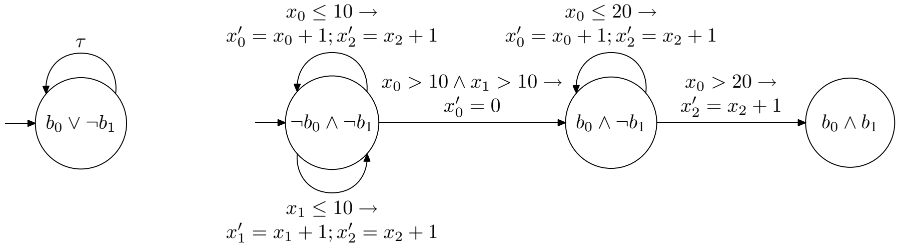
Numerical acceleration can be applied to self-loops where the numerical state evolves while the Boolean state does not: see Fig. 3 for an example and a counterexample. The tool Aspic [12] is based on the enumeration of the Boolean state space which trivially yields a CFG that fulfills this requirement.

Example 1. We will try to infer invariants on the following running example:
$\mathcal{I}(\boldsymbol{b},\boldsymbol{x}) = \neg b_0 \wedge \neg b_1 \wedge x_0{=}0 \wedge x_1{=}0 \wedge x_2{=}0$

$$\mathit{true} \rightarrow \begin{cases} b_0' = b_0 \quad \vee \quad (\neg b_0 \wedge x_0{>}10 \wedge x_1{>}10) \\ b_1' = b_1 \quad \vee \quad (\neg b_1 \wedge x_0{>}20) \\ x_0' = \begin{cases} x_0 + 1 \text{ if } (\neg b_0 \wedge \neg b_1 \wedge x_0{\leq}10 \wedge \beta) \quad \vee \quad (b_0 \wedge \neg b_1 \wedge x_0{\leq}20) \\ 0 \qquad\;\; \text{if } \neg b_0 \wedge \neg b_1 \wedge x_0{>}10 \wedge x_1{>}10 \\ x_0 \qquad\; \text{otherwise} \end{cases} \\ x_1' = \begin{cases} x_1 + 1 \text{ if } \neg b_0 \wedge \neg b_1 \wedge x_1{\leq}10 \wedge \neg\beta \\ x_1 \qquad\; \text{otherwise} \end{cases} \\ x_2' = \begin{cases} x_2 + 1 \text{ if } (\neg b_0 \wedge \neg b_1 \wedge (x_0{\leq}10 \wedge \beta \vee x_1{\leq}10 \wedge \neg\beta)) \quad \vee \quad (b_0 \wedge \neg b_1) \\ x_2 \qquad\; \text{otherwise} \end{cases} \end{cases}$$

The counting patterns of this example (see Fig. 4b) is representative of the production line benchmarks presented in Section 5.

Generating a numerical CFG. At first, one performs a Boolean reachability analysis in order to reduce the state space of interest ($b_0 \vee \neg b_1$ in the case of our running example). Starting from the most simple CFG of the program consisting of a single location with a self-loop (see Fig. 4a), standard techniques are used for (1) *enumerating* the Boolean state space and (2) *simplifying the transitions* by source and destination location using partial evaluation. Afterwards, (3) the *Boolean input variables* are replaced by explicit non-deterministic transitions (see Fig. 4b). This CFG is purely numerical, but the guards of the loop transitions might still be non-convex. Transforming the guard into a minimal DNF and splitting the transition into several transitions, one for each conjunct, yields a CFG with self-loops compatible with the transition scheme of §2.2. A


τ
$b_0 \vee \neg b_1$
$x_0 \leq 10 \rightarrow$
$x'_0 = x_0 + 1; x'_2 = x_2 + 1$
$\neg b_0 \wedge \neg b_1$
$x_1 \leq 10 \rightarrow$
$x'_1 = x_1 + 1; x'_2 = x_2 + 1$
$x_0 > 10 \wedge x_1 > 10 \rightarrow$
$x'_0 = 0$
$x_0 \leq 20 \rightarrow$
$x'_0 = x_0 + 1; x'_2 = x_2 + 1$
$b_0 \wedge \neg b_1$
$x_0 > 20 \rightarrow$
$x'_2 = x_2 + 1$
$b_0 \wedge b_1$

(a) Initial CFG

(b) CFG after Boolean enumeration and removal of the Boolean inputs. (Identity transition functions are implicit.)

Fig. 4. Transformation of the program of Example 1. τ is the global transition. The guards are already convex in the obtained CFG.

single self-loop like in location $b_0 \wedge \neg b_1$ in Fig. 4b can now be "flattened" into a transitive closure transition (cf. Fig. 2).

Multiple self-loops. However, the obtained CFG usually contains multiple self-loops like in location $\neg b_0 \wedge \neg b_1$ in Fig. 4b. In this case a simple "flattening" as in Fig. 2 is not possible: For the fixed point computation we must take into account all sequences of self-loop transitions in this location. Actually, the idempotency of accelerated transitions can be exploited in order reduce these sequences to those where the same transition is never taken twice successively: For the two accelerable loops we have to compute:
$\tau_1^{\otimes}(X) \sqcup \tau_2^{\otimes}(X) \sqcup \tau_2^{\otimes} \circ \tau_1^{\otimes}(X) \sqcup \tau_1^{\otimes} \circ \tau_2^{\otimes}(X) \sqcup \tau_1^{\otimes} \circ \tau_2^{\otimes} \circ \tau_1^{\otimes}(X) \sqcup \tau_2^{\otimes} \circ \tau_1^{\otimes} \circ \tau_2^{\otimes}(X) \sqcup \ldots$
This infinite sequence may not converge, thus in general, widening is necessary to guarantee termination. However, in practice the sequence often converges after the first few elements (see [5]).

The technique implemented in Aspic consists in expanding multiple self-loops into a graph of which the paths represent these sequences, as shown in Fig. 5 in the case of three self-loops, and to solve iteratively the fixed point equations induced by the CFG as sketched in §2.1, using widening if necessary. Moreover, Aspic implements methods to accelerate circuits of length greater than one.

3 Logico-Numerical Abstract Acceleration

Our goal is to exploit abstract acceleration techniques *without resorting to a Boolean state space enumeration* in order to overcome the limitations of current tools (e.g. [12]) w.r.t. the analysis of logico-numerical programs.

In this section we will first discuss some related issues in order to motivate our approach before presenting methods that make abstract acceleration applicable to a CFG, which now may contain loops with operations on both Boolean and numerical variables.

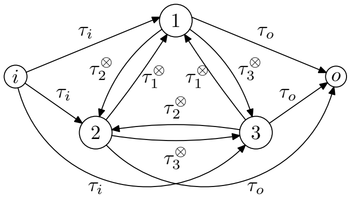


Fig. 5. Computation of three accelerable self-loops τ_1, τ_2 and τ_3. τ_i and τ_o are the incoming resp. outgoing transitions of the location.

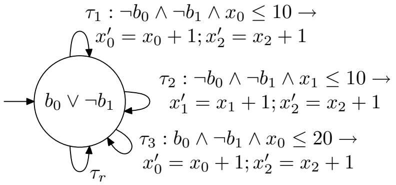

$\tau_1 : \neg b_0 \wedge \neg b_1 \wedge x_0 \leq 10 \rightarrow$
$x_0' = x_0 + 1; x_2' = x_2 + 1$
$\tau_2 : \neg b_0 \wedge \neg b_1 \wedge x_1 \leq 10 \rightarrow$
$x_1' = x_1 + 1; x_2' = x_2 + 1$
$b_0 \vee \neg b_1$
$\tau_3 : b_0 \wedge \neg b_1 \wedge x_0 \leq 20 \rightarrow$
$x_0' = x_0 + 1; x_2' = x_2 + 1$
τ_r

Fig. 6. Acceleration of Ex. 1 in a CFG with a single location: The upper three self-loops are accelerable. The rest of the system is summarized in the transition τ_r where the Boolean equations are not the identity.

3.1 Motivations for Our Approach

A first observation is that identifying self-loops is more complex when Boolean state variables are not fully encoded in the CFG. Indeed, if a symbolic CFG contains a *"syntactic"* self-loop (ℓ, τ, ℓ) with $\tau : g(\boldsymbol{b}, \boldsymbol{x}, \boldsymbol{\xi}) \rightarrow (\boldsymbol{b}, \boldsymbol{x}) = \boldsymbol{f}(\boldsymbol{b}, \boldsymbol{x}, \boldsymbol{\xi})$, there is an *"effective"* self-loop only for those Boolean states $\boldsymbol{b} \in \varphi_l$ such that $g(\boldsymbol{b}, \boldsymbol{x}, \boldsymbol{\xi}) \wedge \boldsymbol{b} = \boldsymbol{f}^b(\boldsymbol{b}, \boldsymbol{x}, \boldsymbol{\xi})$ is satisfiable[1]. For instance, the self-loop around location $\neg b_1$ in Fig. 3 is not an "effective" self-loop.

This observation also applies to circuits, where *numerical inputs have to be duplicated*: If there is a circuit (ℓ, τ_1, ℓ') and (ℓ', τ_2, ℓ) with $\tau_i : g_i(\boldsymbol{s}, \boldsymbol{\xi}) \rightarrow \boldsymbol{s}' = \boldsymbol{f_i}(\boldsymbol{s}, \boldsymbol{\xi})$ for $i = 1, 2$, the composed transition has the form $\tau : g(\boldsymbol{s}, \boldsymbol{\xi}, \boldsymbol{\xi}') \rightarrow \boldsymbol{s}'' = \boldsymbol{f}(\boldsymbol{s}, \boldsymbol{\xi}, \boldsymbol{\xi}')$. This strongly limits in practice the length of circuits that can be reduced to self-loops and accelerated. In this paper, we will not deal with such circuits, and we consider only self-loops.

We give a definition for a logico-numerical self-loop which can be accelerated by the known methods, because the Boolean part of the transition function is the identity:

Definition 2 (Accelerable logico-numerical transition). *A transition τ is said to be accelerable if it has the form* $g^b(\boldsymbol{b}, \boldsymbol{\beta}) \wedge g^x(\mathcal{C}) \rightarrow \begin{pmatrix} \boldsymbol{b}' \\ \boldsymbol{x}' \end{pmatrix} = \begin{pmatrix} \boldsymbol{b} \\ \boldsymbol{a}(\boldsymbol{x}, \boldsymbol{\xi}) \end{pmatrix}$, *where* $g^x(\mathcal{C}) \rightarrow \boldsymbol{x}' = \boldsymbol{a}(\boldsymbol{x}, \boldsymbol{\xi})$ *is accelerable according to* §*2.2*.

A naive approach to our problem could be to partition the system into sufficiently many locations, until we get self-loops that correspond to Def. 2. This approach is simple-minded for two reasons: (i) There might be no such Boolean states in the program at all; (ii) in the case of Fig. 3, simply ignoring the Boolean variable b_2 would make the (syntactic) self-loop accelerable without impacting the precision. More generally, it may pay off to slightly abstract the behaviour of self-loops in order to benefit from precise acceleration techniques.

Another important remark is that we do not necessarily need to partition the system into locations to apply acceleration: it is sufficient to decompose the self-loops: Starting from the basic CFG with a single location and a single self-loop,

[1] We assume here that Bool. inputs β have been encoded by non-determinism, see §2.3.

we could split the loop into loops where the numerical transition function can be accelerated and the Boolean transition is the identity and a last loop where this is not the case. Fig. 6 shows the result of the application of this idea to our running example of Fig. 4.

This allows us to *separate the issue of accelerating self-loops in a symbolic CFG*, addressed in this section, *from the issue of finding a suitable CFG*, addressed in §4. We will use a dedicated partitioning technique to find an appropriate CFG in order to render effective our logico-numerical acceleration method.

3.2 Decoupling Numerical and Boolean Transition Functions

We consider self-loops (ℓ, τ, ℓ) with $\tau : \mathcal{A}(\boldsymbol{s}, \boldsymbol{i}) \rightarrow \begin{pmatrix} \boldsymbol{b}' \\ \boldsymbol{x}' \end{pmatrix} = \begin{pmatrix} \boldsymbol{f}^b(\boldsymbol{s}, \boldsymbol{i}) \\ \boldsymbol{f}^x(\boldsymbol{s}, \boldsymbol{i}) \end{pmatrix}$. We use the abstractions $\wp(E) = \wp(\mathbb{B}^m \times \mathbb{R}^n) \underset{\pi}{\overset{id}{\longleftrightarrow}} \wp(\mathbb{B}^m) \times \wp(\mathbb{R}^n) \underset{\alpha}{\overset{id}{\longleftrightarrow}} A = \wp(\mathbb{B}^m) \times Pol(\mathbb{R}^n)$ discussed in §2.1, where π is the function that approximates a set $S \in E$ by a Cartesian product, *e.g.* $\pi((B_1 \times X_1) \cup (B_2 \times X_2)) = (B_1 \cup B_2) \times (X_1 \cup X_2)$. If τ is accelerable in the sense of abstract acceleration, then $\pi \circ \tau^* \subseteq \tau^\otimes$.

Our logico-numerical abstract acceleration method relies on *decoupling* the numerical and Boolean parts of the transition function τ with

$$\tau_b : \mathcal{A}(\boldsymbol{s}, \boldsymbol{i}) \rightarrow \begin{pmatrix} \boldsymbol{b}' \\ \boldsymbol{x}' \end{pmatrix} = \begin{pmatrix} f^b(\boldsymbol{s}, \boldsymbol{i}) \\ \lambda(\boldsymbol{s}, \boldsymbol{i}).\, \boldsymbol{x} \end{pmatrix} \text{ and } \tau_x : \mathcal{A}(\boldsymbol{s}, \boldsymbol{i}) \rightarrow \begin{pmatrix} \boldsymbol{b}' \\ \boldsymbol{x}' \end{pmatrix} = \begin{pmatrix} \lambda(\boldsymbol{s}, \boldsymbol{i}).\, \boldsymbol{b} \\ f^x(\boldsymbol{s}, \boldsymbol{i}) \end{pmatrix}.$$

We can approximate τ^* as follows:

Proposition 1. $\tau^* \subseteq (\pi \circ \tau_b \circ \tau_x^*)^*$.

See [13] for details of the proof. Briefly, we prove first $\tau \subseteq \pi \circ \tau_b \circ (id \cup \tau_x)$. Then, with $(id \cup \tau_x) \subseteq \tau_x^*$ we conclude $\tau^* \subseteq (\pi \circ \tau_b \circ \tau_x^*)^*$.

Now, we assume that τ_x is accelerable in the sense of Def. 2, which means that $\mathcal{A}(\boldsymbol{s}, \boldsymbol{i}) = g^b(\boldsymbol{b}, \boldsymbol{\beta}) \wedge g^x(\boldsymbol{x}, \boldsymbol{\xi})$ and $\boldsymbol{f}^x(\boldsymbol{s}, \boldsymbol{i}) = \boldsymbol{a}(\boldsymbol{x}, \boldsymbol{\xi})$. By applying Prop. 1, we obtain that $(\pi \circ \tau_b \circ \tau_x^\otimes)^*$ is a sound approximation of τ^*. Although we could prove that the involved Kleene iteration is bounded and converges without applying widening, there exists a more efficient alternative in which numerical and Boolean parts are computed in sequence, so that numerical acceleration is applied only once.

Proposition 2. *If τ_x is accelerable, then*

(1) $(\pi \circ \tau_b)^* \circ \pi \circ \tau_x^* \circ \pi$ *is idempotent, and*
(2) $\tau^* \subseteq (\pi \circ \tau_b)^* \circ \pi \circ \tau_x^* \circ \pi$

See [13] for details of the proof. The intuition for (1) is the following: If the guard $g^x \wedge g^b$ is satisfied, *i.e.* the transition can be taken, we saturate the numerical dimensions first; then we saturate the Boolean ones. The point is now, that the application of τ_b does not enable "more" behavior of the numerical variables. Thus, re-applying the function has no effect.
Then we can prove (2): from Prop. 1 follows $\tau^* \subseteq (\pi \circ \tau_b \circ \tau_x^*)^* = ((\pi \circ \tau_b)^* \circ \tau_x^*)^* \subseteq$

$((\pi \circ \tau_b)^* \circ \pi \circ \tau_x^* \circ \pi)^* = (\pi \circ \tau_b)^* \circ \pi \circ \tau_x^* \circ \pi$ (for the last step, we use (1) and the fact that the function includes the identity).

The following theorem implements Prop. 2 in the abstract domain A. We use the notation $h = f \downarrow X$ ("f partially evaluated on the convex polyhedron X"), to denote any (simpler) formula h such that $X(\boldsymbol{x}, \boldsymbol{\xi}) \Rightarrow (h(\boldsymbol{b}, \boldsymbol{\beta}, \boldsymbol{\mathcal{C}}) = f(\boldsymbol{b}, \boldsymbol{\beta}, \boldsymbol{\mathcal{C}}))$.

Theorem 1. *If a transition τ is such that τ_x is accelerable, then τ^* can be approximated in A with* $\tau^\otimes : \begin{array}{ccc} A & \to & A \\ (B, X) & \mapsto & \left(\left(\tau_b^b\left[X^\otimes\right]\right)^* (B)\ ,\ X^\otimes\right) \end{array}$
where
- $X^\otimes = (\tau_x^x)^\otimes(X)$
- $(\tau_x^x)^\otimes$ *is the abstract acceleration of* $\tau_x^x : g^x(\boldsymbol{x}, \boldsymbol{\xi}) \to \boldsymbol{x}' = \boldsymbol{a}(\boldsymbol{x}, \boldsymbol{\xi})$
- $\tau_b^b[X](B) = \left\{ \left(\boldsymbol{f}^b \downarrow (X \sqcap g^x)\right)(\boldsymbol{b}, \boldsymbol{\beta}, \boldsymbol{\mathcal{C}}) \mid \boldsymbol{b} \in B \wedge g^b(\boldsymbol{b}, \boldsymbol{\beta}) \right\}$
- $(\tau_b^b[X])^*(B) = \mathit{lfp}(\lambda B' \,.\, B \cup \tau_b^b[X](B'))$.

Moreover, $(\tau_b^b[X])^*$ *(and thus* $\tau^\otimes$*) can be computed in bounded time as the least fixed point of a monotonic function in the finite lattice* $\wp(\mathbb{B}^m)$*.*

In other words, we compute the transitive closure $X^\otimes$ of τ_x using numerical abstract acceleration and saturate τ_b partially evaluated over $X^\otimes$.

Discussion. At the first glance the approximations induced by this partial decoupling seem to be rather coarse. However, it is not really the case in our context for two reasons:

1. The correlations between Boolean and numerical variables that are lost by our method are mostly not representable in the abstract domain A anyway. For example, consider the loop $x \leq 4 \to (b' = \neg b; x' = x+1)$, where b could be the least significant bit of a binary counter for instance: starting from $(b, x) \in \{(\mathit{true}, 0)\}$ the exact reachable set is $\{\mathit{true}\} \times \{0, 2, 4\} \cup \{\mathit{false}\} \times \{1, 3, 5\}$; its abstraction in A is $\{\top\} \times \{0 \leq x \leq 5\}$. Hence, this information will also be lost in a standard analysis merely relying on widening. Yet, due to numerical acceleration we can even expect a better precision with our method.
2. We will apply this method to CFGs (see §4) in which the Boolean states defining a location exhibit the same numerical behavior and thus, decoupling is supposed not to seriously affect the precision.

Until now we studied the case of a single self-loop. In the presence of multiple self-loops we expand the graph in the same way as with purely numerical transitions, *e.g.* as shown in Fig. 5, and we apply Thm. 1 to each loop. As in the purely numerical case, widening must be applied in order to guarantee convergence.

Example 2. We give the results obtained for our running example: Analyzing the enumerated CFG in Fig. 4b using abstract acceleration gives $0 \leq x_0 \leq 21 \wedge$ $0 \leq x_1 \leq 11 \wedge\ x_0 + x_1 \leq x_2 \leq 44$ bounding all variables[2]. Analyzing the system on a CFG with a single location using decoupling and abstract acceleration still bounds two variables $(0 \leq x_0 \leq 21 \wedge 0 \leq x_1 \leq 11 \wedge x_0 + x_1 \leq x_2)$, whereas, even on

[2] Over-approximated result: the actual polyhedron has more constraints.

the enumerated CFG standard analysis does not find any upper bound at all: $0 \leq x_0 \wedge 0 \leq x_1 \wedge x_0 + x_1 \leq x_2$.

3.3 Decoupling Accelerable from Non-accelerable and Boolean Transition Functions

Theorem 1 applies only if the numerical transition functions are accelerable. If this is not the case, we can reuse the idea of Prop. 1, but now by decoupling the accelerable numerical functions from Boolean and non-accelerable numerical functions:

$$\tau_a : \mathcal{A}(\boldsymbol{s}, \boldsymbol{i}) \rightarrow \begin{pmatrix} \boldsymbol{b}' \\ \boldsymbol{x}'_n \\ \boldsymbol{x}'_a \end{pmatrix} = \begin{pmatrix} \lambda(\boldsymbol{s}, \boldsymbol{i}).\, \boldsymbol{b} \\ \lambda(\boldsymbol{s}, \boldsymbol{i}).\, \boldsymbol{x}_n \\ \boldsymbol{a}(\boldsymbol{x}, \boldsymbol{\xi}) \end{pmatrix} \;, \; \tau_{n,b} : \mathcal{A}(\boldsymbol{s}, \boldsymbol{i}) \rightarrow \begin{pmatrix} \boldsymbol{b}' \\ \boldsymbol{x}'_n \\ \boldsymbol{x}'_a \end{pmatrix} = \begin{pmatrix} \boldsymbol{f}^b(\boldsymbol{s}, \boldsymbol{i}) \\ \boldsymbol{f}^n(\boldsymbol{s}, \boldsymbol{i}) \\ \lambda(\boldsymbol{s}, \boldsymbol{i}).\, \boldsymbol{x}_a \end{pmatrix}$$

Proposition 3. $\tau^* \subseteq (\pi \circ \tau_{n,b} \circ \tau_a^*)^* \subseteq (\pi \circ \tau_{n,b} \circ \tau_a^{\otimes})^*$

However, Prop. 2 does not apply any more, because the function τ_a depends on non-accelerated numerical variables updated by $\tau_{n,b}$. Moreover, widening is required because $\tau_{n,b}^*$ is not guaranteed to converge in a bounded number of iterations.

3.4 Using Inputization Techniques

Inputization (see [14] for instance) is a technique that treats state variables as input variables. This method is useful to cut dependencies. For example, it can be employed to reduce $\big((\pi \circ \tau_b)^* \circ \pi\big)$ to $\big(\pi \circ \tau'_b \circ \pi\big)$ in Prop. 1, where τ'_b is computed by inputizing in τ_b the Boolean state variables having a transition function which is neither the identity nor constant.

Example 3. The loop τ_b can be approximated by the transition τ'_b where β_0 and β_2 correspond to b_0 and b_2 manipulated as Boolean inputs:

$$\tau_b : \left| \begin{array}{l} b'_0 = \neg b_0 \\ b'_1 = b_1 \\ b'_2 = b_2 \wedge x \geq 0 \end{array} \right. \qquad \tau'_b : \left| \begin{array}{l} b'_0 = \beta_0 \\ b'_1 = b_1 \\ b'_2 = \beta_2 \wedge x \geq 0 \end{array} \right.$$

Our experiments show that this technique is quite useful: the speed-up gained by removing loops often pays off in comparison to the approximations it brings about.

4 Partitioning Techniques for Logico-Numerical Acceleration

The logico-numerical acceleration method described in the previous section can be applied to any CFG. However, in order to make it effective we apply it to a CFG obtained by a partitioning technique that aims at alleviating the impact of decoupling on the precision. This section proposes such partitioning techniques that generate CFGs in which the Boolean states that exhibit the same numerical behavior are grouped in the same locations, so that it is likely that the numerical transition functions in loops do not depend on Boolean state variables.

Basic technique. In order to implement this idea we generate a CFG that is characterized by the following equivalence relation:

Definition 3. *(Boolean states with same set of guarded numerical actions)*

$$\boldsymbol{b}_1 \sim \boldsymbol{b}_2 \Leftrightarrow \begin{cases} \forall \boldsymbol{\beta}_1, \mathcal{C} : \mathcal{A}(\boldsymbol{b}_1, \boldsymbol{\beta}_1, \mathcal{C}) \Rightarrow \\ \qquad \exists \boldsymbol{\beta}_2 : \mathcal{A}(\boldsymbol{b}_2, \boldsymbol{\beta}_2, \mathcal{C}) \wedge \boldsymbol{f}^x(\boldsymbol{b}_1, \boldsymbol{\beta}_1, \mathcal{C}) = \boldsymbol{f}^x(\boldsymbol{b}_2, \boldsymbol{\beta}_2, \mathcal{C}) \\ \quad \textit{and vice versa} \end{cases}$$

The intuition of this heuristics is to make equivalent the Boolean states that can execute the same set of *numerical actions*, guarded by the same numerical constraints.

Example 4. We illustrate the application of this method to Example 1. We first factorize the numerical transition functions by actions:

$$(x_0', x_1', x_2') = \begin{cases} (x_0+1, \; x_1 \;, x_2+1) & \text{if } (\neg b_0 \wedge \neg b_1 \wedge x_0 \le 10) \;\vee\; (b_0 \wedge \neg b_1 \wedge x_0 \le 20) \\ (\; x_0 \;, x_1+1, x_2+1) & \text{if } \neg b_0 \wedge \neg b_1 \wedge x_1 \le 10 \\ (\; 0 \;, \; x_1 \;, \; x_2 \;) & \text{if } \neg b_0 \wedge \neg b_1 \wedge x_0 > 10 \wedge x_1 > 10 \\ (\; x_0 \;, \; x_1 \;, x_2+1) & \text{if } b_0 \wedge \neg b_1 \wedge x_0 > 20 \\ (\; x_0 \;, \; x_1 \;, \; x_2 \;) & \text{otherwise} \end{cases}$$

Then by applying Def. 3 we get the equivalence classes $\{\neg b_0 \wedge \neg b_1, b_0 \wedge \neg b_1, b_0 \wedge b_1\}$: the obtained CFG is the one of Fig. 4b.

In the worst case, as in Ex. 4 above, a different set of actions can be executed in each Boolean state, thus the Boolean states will be enumerated. In the other extreme case in all Boolean states the same set of actions can be executed, which induces a single equivalence class. Both cases are unlikely to occur in larger, real systems.

From an algorithmic point of view, we represent all our functions with BDDs and MTBDDs [15], and we proceed as follows: We factorize the numerical transition functions by the numerical actions (trivial with MTBDDs):

$$\boldsymbol{f}^x(\boldsymbol{b}, \boldsymbol{\beta}, \mathcal{C}, \boldsymbol{x}, \boldsymbol{\xi}) = \bigvee_{1 \le i \le m} \big(g_i(\boldsymbol{b}, \boldsymbol{\beta}, \mathcal{C}) \rightarrow \boldsymbol{a}_i^x(\boldsymbol{x}, \boldsymbol{\xi})\big)$$

Then we eliminate the Boolean inputs β, and we decompose the results into

$$(\exists \boldsymbol{\beta} : g_i(\boldsymbol{b}, \boldsymbol{\beta}, \mathcal{C})) = \bigvee_{1 \le j \le n_i} g_{ij}^b(\boldsymbol{b}) \wedge g_{ij}^x(\mathcal{C})$$

where $g_{ij}^x(\mathcal{C})$ may be non-convex. The equivalence relation $\sim$ of Def. 3 can be reformulated as

$$\boldsymbol{b}_1 \sim \boldsymbol{b}_2 \Leftrightarrow \forall i \forall j : g_{ij}^b(\boldsymbol{b}_1) \Leftrightarrow g_{ij}^b(\boldsymbol{b}_2).$$

This last formulation reflects the fact that in the resulting CFG the numerical function $\boldsymbol{f}^x$ specialized on a location ℓ does not depend any more on $\boldsymbol{b}$. Hence, the information loss is supposed to be limited.

Reducing the size of the partition. An option for having a less discriminating equivalence relation is to make equivalent the Boolean states that can execute the same set of numerical actions *regardless of the numerical constraints guarding them.*

Definition 4. *(Boolean states with same set of numerical actions)*

$$\boldsymbol{b}_1 \approx \boldsymbol{b}_2 \Leftrightarrow \begin{cases} \forall \boldsymbol{\beta}_1, \mathcal{C}_1 : \mathcal{A}(\boldsymbol{b}_1, \boldsymbol{\beta}_1, \mathcal{C}_1) \Rightarrow \\ \qquad \exists \boldsymbol{\beta}_2, \mathcal{C}_2 : \mathcal{A}(\boldsymbol{b}_2, \boldsymbol{\beta}_2, \mathcal{C}_2) \wedge \boldsymbol{f}^x(\boldsymbol{b}_1, \boldsymbol{\beta}_1, \mathcal{C}_1) = \boldsymbol{f}^x(\boldsymbol{b}_2, \boldsymbol{\beta}_2, \mathcal{C}_2) \\ \quad \textit{and vice versa} \end{cases}$$

We clearly have $\sim \subseteq \approx$. For example, if we have two guarded actions $b \wedge x \leq 10 \rightarrow x' = x+1$ and $\neg b \wedge x \leq 20 \rightarrow x' = x+1$, $\sim$ will separate the Boolean states satisfying resp. b and $\neg b$, whereas $\approx$ will keep them together.

Another option is to consider only a subset of the numerical actions, that is, we ignore the transition functions of some numerical variables in Defs. 3 or 4. One can typically focus only on variables involved in the property. According to our experiments, this method is very efficient, but it relies on manual intervention.

5 Experimental Evaluation

Our experimentation tool nbACCel implements the proposed methods on the basis of the logico-numerical abstract domain library BddApron [16].

Benchmarks. Besides some small, but difficult benchmarks, we used primarily benchmarks that are simulations of *production lines* as modeled with the library Quest for the LCM language[3] (see Fig. 1) for evaluating scalability. These models consist of building blocks like sources, buffers, machines, routers for splitting and combining flows of material and sinks, that synchronize via handshakes. The properties we want to prove depend on numerical variables, *e.g.* (1) maximal throughput time of the first element passing the production line, or (2) minimal throughput of the production line. Inputs could serve modeling nondeterministic processing and arrival times, but we did not choose benchmarks with numerical inputs in order to enable a comparison with Aspic [12].

Results. We compared our tool nbACCel with NBac [4] and Aspic. The results are summarized in Table 1. The tools where launched with the default options; for nbACCel we use the partitioning heuristics of Def. 4 and the inputization technique of §3.4. We do not need the technique of §3.3 for our examples.

Discussion. The experimental comparison gives evidence about the advantages of abstract acceleration, but also some potential for future improvement:

- nbACCel can prove a lot of examples where NBac fails: this is due to the fact that abstract acceleration improves precision, especially in nested loops where the innermost loop can be "flattened", which makes it possible to recover more information in descending iterations.
- nbACCel seems to scale better than NBac: First, the idempotency of abstract acceleration reduces the number of iterations and fixed point checks. Second, our heuristics generates a partition that is well-suited for analysis – though, for some of the larger benchmarks, *e.g.* LCM quest 4-1, the dynamic partitioning of NBac starts to pay off, whereas our static partition is more fine-grained than necessary, which makes us waste time during analysis.

[3] `http://www.3ds.com`

Table 1. Experimental comparison between Aspic, nbACCel and Nbac

		Aspic		nbACCel		Nbac	
	vars	size	time	size	time	size	time
Gate 1	4/4/2	7	?	5	**0.73**	24	?
Escalator 1	5/4/2	12	**0.14 (0.04)**	9	0.49	22	?
Traffic 1	4/6/0	18	**0.14 (0.01)**	16	0.19	5	3.49
Traffic 2	4/8/0	18	?	16	**0.35**	28	?
LCM Quest 0a-1	7/2/0	7	**0.04 (0.01)**	5	**0.04**	5	0.05
LCM Quest 0a-2	7/3/0	6	**0.05 (0.01)**	4	**0.05**	8	0.19
LCM Quest 0b-1	10/3/0	19	**0.08 (0.01)**	12	**0.08**	9	?
LCM Quest 0b-2	10/4/0	17	**0.09 (0.01)**	11	0.20	33	?
LCM Quest 0c-1	15/4/0	28	0.17 (0.01)	16	**0.16**	8	0.86
LCM Quest 0c-2	15/5/0	25	**0.20 (0.05)**	14	0.24	50	14.8
LCM Quest 1-1	16/5/0	114	1.99 (0.48)	42	**0.92**	6	2.45
LCM Quest 1-2	16/6/0	100	?	34	?	>156	>
LCM Quest 1b-1	16/5/0	55	0.92 (0.04)	29	**0.37**	15	?
LCM Quest 1b-2	16/5/0	45	0.76 (0.12)	23	**0.47**	61	?
LCM Quest 2-1	17/6/0	247	c	82	**7.84**	9	12.8
LCM Quest 2-2	17/7/0	198	>	62	?	>76	>
LCM Quest 3-1	25/5/0	483	26.5 (14.4)	58	8.49	12	**3.76**
LCM Quest 3-2	25/6/0	481	c	54	?	>1173	>
LCM Quest 3b-1	26/6/0	1724	>	170	43.8	14	**19.1**
LCM Quest 3b-2	26/7/0	1710	>	162	>	>32	>
LCM Quest 3c-1	26/6/0	1319	>	130	**34.2**	9	?
LCM Quest 3c-2	26/7/0	1056	c	98	>	>70	>
LCM Quest 3d-1	26/6/0	281	>	81	**5.43**	49	?
LCM Quest 3d-2	26/7/0	266	c	73	?	446	?
LCM Quest 3e-1	27/7/0	638	>	140	**20.6**	49	?
LCM Quest 3e-2	27/8/0	514	>	110	**6.46**	>28	>
LCM Quest 4-1	27/7/0	4482	>	386	186	9	**50.1**
LCM Quest 4-2	27/8/0	3586	>	290	>	>6	>

vars	: Boolean state variables / numerical state variables / Boolean inputs
size	: number of locations of the CFG
time	: in seconds (Aspic: total time (time for analysis))
?	: "don't know" (property not proved)
>	: timed out after 600s
c	: out of memory or crashed

(Benchmarks on `http://pop-art.inrialpes.fr/people/schramme/nbaccel/`)

- Once provided with an enumerated CFG, Aspic is very fast on the smaller benchmarks. However, the current version (3.1) cannot deal with CFGs larger than a few hundred locations. We were surprised that some of the small examples were not proven by Aspic. We suspect that this is due to some information loss in widening.
- The analysis using logico-numerical acceleration proved twice as many benchmarks and turned out to be 20% faster than a standard analysis of the same CFG with widening with delay 2 and two descending iterations.
- Applying the more refined partition of Def. 3 to our benchmarks had only a minor influence on performance and precision, and not applying inputization had no impact on the verification of properties, but it slowed down the analysis by 25% on average.
- Generally, for the benchmarks LCM quest 1 to 4 property 2 was not proved by the tools. Here, the combination of our heuristics with dynamic partitioning for further refining the critical parts of the CFG could help.

6 Conclusion and Related Work

We propose techniques for accelerating logico-numerical transitions, that allow us to benefit from the precision gain by numerical abstract acceleration as used in the tool Aspic, while tackling the Boolean state space explosion problem encountered when analysing logico-numerical programs. Experimentally, our tool nbACCel is often able to prove properties for the larger benchmarks, unlike the two other tools we tested – and this on CFGs that are ten times smaller than the CFGs obtained by enumeration of the reachable Boolean state space. Although our method is based on the partial decoupling of the Boolean and numerical transitions, the experiments confirm our intuition that our method generally improves the precision. We attribute this to the following observations: first, numerical abstract acceleration reduces the need for widening; second, the information that we might lose by decoupling would often not be captured by the abstract domain anyway; and at last, the CFG obtained by our partitioning method particularly favors the application of our logico-numerical acceleration method.

This work raises interesting perspectives: Regarding abstract acceleration, the acceleration of multiple self-loops deserves additional investigation in relation with partitioning techniques. Concerning partition refinement, the combination of our approach with dynamic partitioning *à la* [4] seems to be worth pursuing. In particular partitioning according to numerical constraints is mandatory for proving properties relying on non-convex inductive invariants. Such improvements should allow to tackle a wider range of benchmarks.

Related Work. To our knowledge there is no work about the application of abstract acceleration to logical-numerical data-flow programs, but there is work on related methods that we tailored to fit our purpose. In §2 we already discussed in detail the concepts of abstract acceleration [1,11], on which our work is based, and that we extended in [6].

Jeannet [4] uses in the tool Nbac partitioning heuristics that are based on the property being analyzed in order to cut paths between initial and bad states. The tool interleaves partitioning steps with analysis (*dynamic partitioning*), thus the "dangerous" state space is reduced in each step. Bouajjani et al. [17] describe a partition refinement algorithm for the Lustre compiler using bisimulation. We think that we could exploit it to refine our CFG, when we fail to prove the property.

Alternative approaches for verifying properties about data-flow programs rely on bounded model-checking or k-induction techniques, which both exploit the efficiency of modern SMT solvers. Hagen and Tinelli [18] describe the application of these two approaches to the verification of Lustre programs. Another example is the HySat tool [19], a bounded model-checker for hybrid systems with piecewise linear behavior – our methods allow to analyze discretizations of such systems. HySat relies on the integration of linear constraint solving with SAT solving. The interesting point is that they deal implicitly with large Boolean control structures by encoding them into linear pseudo-Boolean constraints.

References

1. Gonnord, L., Halbwachs, N.: Combining widening and acceleration in linear relation analysis. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 144–160. Springer, Heidelberg (2006)
2. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Principles of Programming Languages, POPL 1977, pp. 238–252. ACM Press, New York (1977)
3. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Principles of Programming Languages, POPL 1978, pp. 84–97. ACM Press, New York (1978)
4. Jeannet, B.: Dynamic partitioning in linear relation analysis. application to the verification of reactive systems. Formal Methods in System Design 23, 5–37 (2003)
5. Bardin, S., Finkel, A., Leroux, J., Petrucci, L.: Fast: acceleration from theory to practice. Software Tools for Technology Transfer 10, 401–424 (2008)
6. Schrammel, P., Jeannet, B.: Extending abstract acceleration to data-flow programs with numerical inputs. In: Numerical and Symbolic Abstract Domains, NSAD 2010. ENTCS, vol. 267, pp. 101–114 (2010)
7. Jeannet, B.: Partitionnement Dynamique dans l'Analyse de Relations Linéaires et Application à la Vérification de Programmes Synchrones. Thèse de doctorat, Grenoble INP (2000)
8. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
9. Coudert, O., Berthet, C., Madre, J.C.: Verification of synchronous sequential machines based on symbolic execution. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407. Springer, Heidelberg (1990)
10. Bourdoncle, F.: Efficient chaotic iteration strategies with widenings. In: Pottosin, I.V., Bjorner, D., Broy, M. (eds.) FMP&TA 1993. LNCS, vol. 735, pp. 128–141. Springer, Heidelberg (1993)
11. Gonnord, L.: Accélération abstraite pour l'amélioration de la précision en Analyse des Relations Linéaires. Thèse de doctorat, Université Joseph Fourier, Grenoble (2007)
12. Gonnord, L.: The ASPIC tool: Accelerated symbolic polyhedral invariant computation (2009), http://laure.gonnord.org/pro/aspic/aspic.html
13. Schrammel, P., Jeannet, B.: Logico-numerical abstract acceleration and application to the verification of data-flow programs. Technical Report 7630, INRIA (2011)
14. Bres, Y., Gérard Berry, A.B., Sentovich, E.M.: State abstraction techniques for the verification of reactive circuits. In: Designing Correct Circuits, DCC 2002 (2002)
15. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. on Computers 35 (1986)
16. Jeannet, B.: Bddapron: A logico-numerical abstract domain library (2009), http://pop-art.inrialpes.fr/~bjeannet/bjeannet-forge/bddapron/
17. Bouajjani, A., Fernandez, J.C., Halbwachs, N.: Minimal model generation. In: Clarke, E., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 197–203. Springer, Heidelberg (1991)
18. Hagen, G., Tinelli, C.: Scaling up the formal verification of Lustre programs with SMT-based techniques. In: Formal Methods in Computer-Aided Design, FMCAD 2008. IEEE, Los Alamitos (2008)
19. Fränzle, M., Herde, C.: Hysat: An efficient proof engine for bounded model checking of hybrid systems. Formal Methods in System Design 30, 179–198 (2007)

Invisible Invariants and Abstract Interpretation

Kenneth L. McMillan[1] and Lenore D. Zuck[2]

[1] Microsoft Research, Redmond, WA
kenmcmil@microsoft.com
[2] University of Illinois at Chicago, Chicago, IL
lenore@cs.uic.edu

Abstract. The method of Invisible Invariants provides a way to verify safety properties of infinite parameterized classes of finite-state systems using finite-state model checking techniques. This paper looks at invisible invariants from the point of view of abstract interpretation. Viewed in this way, the method suggests a generic strategy for computing abstract fixed points in the case where the best abstract transformer has a high computational cost. This strategy requires only that we can reasonably segregate the infinite concrete state space into finite subsets of increasing size or complexity. We observe that in domains for which the computation of the best abstract transformer may require an exponential number of calls to a theorem prover, we can sometimes reduce the number of theorem prover calls to just one, without sacrificing accuracy.

1 Introduction

The method of Invisible Invariants provides a way to verify safety properties of infinite parameterized classes of finite-state systems using finite-state model checking techniques [ZP04, BFPZ05]. The method applies to systems of N processes, $P_1 || \cdots || P_N$, where N is an unknown natural number, and such that the system is finite-state for any fixed value of N. In a typical application, one chooses a fixed value for the parameter N and uses symbolic model checking to compute the strongest inductive invariant (reachable states) of this finite-state system. This set of states is then *projected* onto, say, two arbitrary processes P_i and P_j. This produces a set of process state pairs S_{ij}. This relation is then *generalized* over all pairs of processes to obtain a candidate invariant for the parameterized system $\forall i, j \in 1 \ldots N.\ S_{ij}$. In practice, we compute this generalization for only a fixed value of N. If this finite set of states is inductive for sufficiently large N, we can conclude using a small model result that it must be invariant for any N. Since the actual invariant is never seen by the user of this method, it is said to be an "invisible" invariant.

In this paper, we will look at the invisible invariants method from the point of view of abstract interpretation [CC77]. That is, we will define a single concrete domain corresponding to the entire parameterized class of systems. We can then define an abstract domain $\mathcal{L}$ corresponding to the set of possible invisible invariants. Roughly speaking, our abstraction mapping α corresponds to

E. Yahav (Ed.): SAS 2011, LNCS 6887, pp. 249–262, 2011.

the "projection" operation, while the concretization γ corresponds to "generalization". Since our γ is conjunctive, we have a best abstract transformer $\tau^\sharp$. The invisible invariants method, when it succeeds, computes exactly the least fixed point of this best transformer.

The interesting thing about this from the point of view of abstract interpretation is that we have computed an over-approximation (the least fixed point of $\tau^\sharp$) from an under-approximation (the reachable states for a fixed value of N). Somehow we have computed the fixed point of the abstract transformer before ever actually applying it. The fact that in practice this approach often does converge to the abstract least fixed point suggests an alternative approach to computing abstract fixed points that might greatly reduce the computational cost when the abstract transformer is computationally expensive. The only requirement for this is that we have a meaningful way to divide the infinite concrete state space into finite subsets of increasing size or complexity. For example, instead of systems of N finite-state processes, we might consider (finite-data) sequential programs with heaps of N objects.

The result is a nested fixed-point computation strategy. The inner fixed point is a purely finite-state one, computed for a fixed value of N. The next level up alternates these fixed points with an analog of the project-and-generalize operation (again for a fixed value of N). At the highest level, these fixed points are alternated with applications of the true $\tau^\sharp$. In the best case, we may obtain a fixed point of $\tau^\sharp$ without ever applying it. From the point of view of abstract interpretation, we have potentially reduced the cost of computing the abstract fixed point. From the point of view of invisible invariants, we allow the possibility to compute an invariant in the case when the existing approach would fail.

Outline of the Paper. In the next section, we introduce our generic abstract fixed point computation strategy. Then in section 3 we show how the Invisible Invariants method embodies this strategy, using Indexed Predicate Abstraction [Lah04] as our abstract domain. Section 4 describes some experiments applying the strategy to computation of abstract fixed points using Indexed Predicate Abstraction. In section 5 we cover related work.

2 Fixed Point Computation Strategy

We consider an infinite concrete state space $\mathcal{S}$, with a given set of initial states $I \subset \mathcal{S}$ and transition relation $T \subseteq \mathcal{S} \times \mathcal{S}$. Our concrete domain $\mathcal{C}$ is $2^{\mathcal{S}}$ (the lattice of subsets of $\mathcal{S}$). The concrete transformer $\tau : \mathcal{C} \rightarrow \mathcal{C}$ is defined such that its least fixed point is the set of states reachable from I via T, that is, $\tau(S) = I \cup T(S)$.

Our abstract interpretation is defined by a complete abstract lattice $\mathcal{L}$, ordered by $\sqsubseteq$, with $\sqcup$ and $\sqcap$ as lub and glb respectively, and a monotone concretization function $\gamma : \mathcal{L} \rightarrow \mathcal{C}$. We assume that $\mathcal{L}$ is closed under infinite conjunctions (*i.e.* that γ preserves glb's), hence we also have a unique abstraction function $\alpha : \mathcal{C} \rightarrow \mathcal{L}$, where

$$\alpha(S) = \sqcap\{\phi \in \mathcal{L} \mid S \subseteq \gamma(\phi)\},$$

such that γ and α form a Galois connection. This in turn defines a best abstract transformer $\tau^\sharp = \alpha \circ \tau \circ \gamma$ such that l.f.p.$(\tau^\sharp)$ is the strongest inductive invariant of our system expressible in $\mathcal{L}$. Our intention is to compute this least fixed point. However, since the evaluation of $\tau^\sharp$ may be quite costly, we wish to avoid computing it to the extent possible. (See [CC79] and [Cou81].)

Now imagine that our infinite state space is itself a union of an infinite sequence of *finite* subsets $\mathcal{S}_1, \mathcal{S}_2, \ldots$, not necessarily disjoint. To each $\mathcal{S}_N$ we associate a concrete domain $\mathcal{C}_N = 2^{\mathcal{S}_N}$ (the corresponding sub-lattice of $\mathcal{C}$). This situation is depicted in Figure 1. For each sub-lattice, we can also define a corresponding concrete transformer $\tau_N : \mathcal{S}_N \to \mathcal{S}_N$, by simply restricting τ to $\mathcal{S}_N$. That is, for each $S \subseteq \mathcal{S}_N$, $\tau_N(S) = \tau(S) \cap \mathcal{S}_N$. Similarly, we can define restricted concretization and abstraction functions $\gamma_N : \mathcal{L} \to \mathcal{C}_N$ and $\alpha_N : \mathcal{C}_N \to \mathcal{L}$, such that $\gamma_N(\phi) = \gamma(\phi) \cap \mathcal{S}_N$ and $\alpha_N(S) = \alpha(S)$. Finally, we can define restricted best abstract transformers $\tau_N^\sharp = \alpha_N \circ \tau_N \circ \gamma_N$.

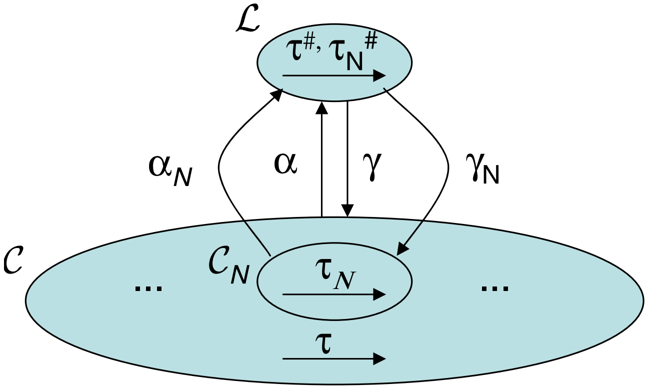


Fig. 1. Division of concrete domain into finite subdomains

It is immediate from the above definitions that the projection of a fixed point of $\tau^\sharp$ is also a fixed point of $\tau_N^\sharp$, that is, l.f.p.$(\tau_N^\sharp) \sqsubseteq$ l.f.p.$(\tau^\sharp)$. Another way to say this is that an inductive invariant of the system is also an inductive invariant of the sub-system. The converse, of course, may not hold. In general, l.f.p.$(\tau_N^\sharp)$ is an under-approximation of l.f.p.$(\tau^\sharp)$.

Despite the fact that it is an under-approximation, $\tau_N^\sharp$ has a distinct advantage. That is, because $\mathcal{S}_N$ is finite, we can compute γ_N directly as a finite set. By composing γ_N, τ_N and α_N, we can compute $\tau_N^\sharp$ exactly, using only finite-state methods. On the other hand, since $\mathcal{S}$ is infinite, we cannot directly compute γ. One approach to computing $\tau^\sharp$ exactly is to use a theorem prover to compute a sequence of approximations to $\tau^\sharp(\phi)$, as is done in predicate abstraction and three-valued shape abstractions [Lah04, RSY04]. A closely related approach is to apply quantifier elimination methods as in [Mon09]. The cost of these methods is usually an exponential number of calls to a theorem prover or decision procedure. Thus it is attractive, to the extent possible, to rely on finite transformers.

To this end, for any monotone transformer ρ over a lattice $\mathcal{L}$, let us define ρ^* to be the transformer that takes a point $\phi \in \mathcal{L}$ and returns the least post fixed point of ρ that is $\sqsupseteq \phi$. That is, $\rho^*(\phi) = \text{l.f.p.}(\lambda X.\ \rho(X) \sqcup \phi)$. We will say that a transformer ρ *under-approximates* a transformer σ when, for all $\phi \in \mathcal{L}$, $\rho(\phi) \sqsubseteq \sigma(\phi)$. We can then show the following simple result about under-approximation:

Theorem 1. *If $\rho, \sigma : \mathcal{L} \rightarrow \mathcal{L}$ are monotone transformers, such that ρ under-approximates σ, then l.f.p.(σ) = l.f.p.$(\sigma \circ \rho^*)$.*

Proof. Suppose $\phi = \text{l.f.p.}(\sigma)$. Since ρ under-approximates σ, we have $\rho(\phi) \sqsubseteq \sigma(\phi) \sqsubseteq \phi$. Therefore ϕ is a fixed point of $\lambda X.\ \rho(X) \sqcup \phi$, necessarily the least. Consequently, $\rho^*(\phi) = \phi$, thus ϕ is a fixed point of $\sigma \circ \rho^*$. Toward a contradiction, suppose that $\phi' \sqsubset \phi$ is a fixed point of $\sigma \circ \rho^*$. Since by definition $\phi' \sqsubseteq \rho^*(\phi')$, it follows by monotonicity of σ that $\sigma(\phi') \sqsubseteq (\sigma \circ \rho^*)(\phi') \sqsubseteq \phi'$. Thus $\phi' \sqsubset \phi$ is a post-fixed point of σ, contradicting $\phi = \text{l.f.p.}(\sigma)$. □

Now consider the following strategy for computing l.f.p.$(\tau^\sharp)$. We begin by computing $\tau_N^{\sharp *}(\bot)$, that is to say, l.f.p.$(\tau_N^\sharp)$. Because $\mathcal{S}_N$ is finite, we can compute this using finite-state methods. If this happens to be a fixed point of $\tau^\sharp$, it is necessarily the least fixed point, so we are done. If not, we can apply $\tau^\sharp$ once, maintaining an under-approximation of the fixed point, and repeat the process, until a fixed point of $\tau^\sharp$ is obtained.

To formalize this idea, let us define a transformer $\pi_N^\sharp = \tau^\sharp \circ \tau_N^{\sharp *}$. To compute this transformer, we take $\tau_N^\sharp$ to a fixed point, then apply $\tau^\sharp$ just once. Since $\tau_N^\sharp$ is an under-approximation of $\tau^\sharp$, by Theorem 1 we have

$$\text{l.f.p.}(\tau^\sharp) = \text{l.f.p.}(\pi_N^\sharp). \tag{1}$$

The advantage of $\pi_N^\sharp$ is that computing its fixed point iteratively might require many fewer iterations, and hence many fewer applications of the expensive $\tau^\sharp$. In fact, in the best case we will have l.f.p.$(\tau_N^\sharp)$ = l.f.p.$(\tau^\sharp)$. In this case computation of the least fixed point of $\pi_N^\sharp$ will terminate in just one iteration. Moreover, applying $\tau^\sharp$ to ϕ, where ϕ is a fixed point, requires only one call to the theorem prover. Thus in the best case, we have reduced the number of theorem prover calls to one.

If we want to take an even more radical stance, we can then use a similar approach to try to reduce the number of computations of γ_N and α_N. This is based on the idea that, for any Galois connection, in addition to the one-step best transformer $\alpha \circ \tau \circ \gamma$, we can also define a many-step best transformer $\alpha \circ \tau^* \circ \gamma$. We can show that iterating these two transformers is equivalent:

Theorem 2. *If $\gamma : \mathcal{L} \rightarrow \mathcal{C}$ and $\alpha : \mathcal{C} \rightarrow \mathcal{L}$ form a Galois connection, and $\tau : \mathcal{C} \rightarrow \mathcal{C}$ is a monotone transformer, then $(\alpha \circ \tau \circ \gamma)^* = (\alpha \circ \tau^* \circ \gamma)^*$.*

Proof. Suppose ϕ is the least post-fixed point of $\alpha \circ \tau \circ \gamma$ that is $\sqsupseteq \psi$. It follows that $\tau(\gamma(\phi)) \subseteq \gamma(\phi)$, thus $\tau^*(\gamma(\phi)) = \gamma(\phi)$. Thus $(\alpha \circ \tau^* \circ \gamma)(\phi) \sqsubseteq \phi$, by the Galois properties. Toward a contradiction, suppose that ϕ' is another post fixed point of $\alpha \circ \tau^* \circ \gamma$ such that $\psi \sqsubseteq \phi' \sqsubset \phi$. Since τ under-approximates τ^*, ϕ' must also be a post-fixed point of $\alpha \circ \tau \circ \gamma$, a contradiction. Thus ϕ is the least post-fixed point of $\alpha \circ \tau^* \circ \gamma$ that is $\sqsupseteq \psi$. □

To compute the least fixed point of $\tau^\sharp_N$, we define a transformer $\rho_N = \alpha_N \circ \tau^*_N \circ \gamma_N$. To compute this transformer, we concretize using γ_N, compute the set of states reachable from this set in sub-system N using finite-state methods, then abstract with α_N. From Theorem 2, we then have:

$$\tau^{\sharp *}_N = \rho^{\sharp *}_N \tag{2}$$

The advantage of $\rho^\sharp_N$ here is again that we may need many fewer iterations to reach a fixed point, and thus we need fewer applications of γ_N and α_N. In the best case, exactly one iteration will be required. Now, using equations (1) and (2), we have:

$$\begin{aligned}
\text{l.f.p.}(\tau^\sharp) &= \text{l.f.p.}(\tau^\sharp \circ \rho^{\sharp *}_N) && (3)\\
&= \text{l.f.p.}(\tau^\sharp \circ (\alpha_N \circ \tau^*_N \circ \gamma_N)^*) && (4)\\
&= (\tau^\sharp \circ (\alpha_N \circ \tau^*_N \circ \gamma_N)^*)^*(\bot) && (5)
\end{aligned}$$

The three nested Kleene stars in this equation correspond to three nested fixed point iterations. The innermost level is a concrete reachability computation. At the intermediate level, we alternate this with abstraction and concretization restricted to sub-system N. At the outermost level, we alternate this with the full abstract best transformer.

We will observe shortly that the Invisible Invariants method corresponds to performing exactly one iteration of the outer and intermediate transformers. Thus, viewed at a high level, it consists of exactly one concrete reachability computation and one application each of α_N, γ_N and $\tau^\sharp$.

What is remarkable is that this is often sufficient to reach the least fixed point of $\tau^\sharp$. The intuition behind this is that the parameter N represents some measure of the size or complexity of a uniformly defined system. As we increase N, at some point the abstract domain loses the ability to distinguish the reachable states of system N from the reachable states of system $N+1$. Thus, even though the concrete reachable states for a fixed system size N under-approximate $\tau^\sharp$ at every step, in the limit the reachable states cannot be distinguished from the abstract fixed point.

Of course this is not always the case. It is possible that the abstract transformer will allow some transition behavior that is qualitatively different from any corresponding concrete transition. This might happen because some important correlation is lost in the abstract domain. However, if this happens infrequently, we might hope that only a small number of outer fixed point iterations will be needed.

Note that at any point in the computation, we also have the option to increase the value of N. We might choose to do this if the counterexample to inductiveness of our current fixed point approximation requires a system larger than N. This would potentially reduce the number of high level iterations, at the cost of a more expensive concrete reachability computation.

A Note on Widening. In the Invisible Invariants method, the abstract domain $\mathcal{L}$ is finite. However, the general approach outlined above applies as well to domains of infinite height, for which a widening might be required to obtain finite convergence. Note that widening is only required in the outermost iterations of equations (1) and (5). In the inner iterations, the concrete domain is finite, thus convergence is guaranteed without widening. As an alternative to extrapolating the sequence obtained by iterating a transformer, it might also be interesting to consider extrapolating the sequence of fixed points obtained as we increase N, that is, l.f.p.$(\tau^{\sharp}_{N})$, l.f.p.$(\tau^{\sharp}_{N+1}), \ldots$

3 Invisible Invariants

In the invisible invariants method, the state space is a space of finite logical structures, mapping variables to values of appropriate sorts. Typically in this structure, there is one distinguished variable $\mathbf{N}$, ranging over the natural numbers, that represents the parameter value. The remaining variables range over various sorts that are dependent on $\mathbf{N}$. For example, we may allow Boolean variables $x_1, \ldots x_a$ ranging over $\{0, 1\}$, scalar variables $y_1, \ldots, y_b$ ranging over $[1 \ldots \mathbf{N}]$ and representing array indices, and array variables $z_1, \ldots, z_c$ that range over maps $[1 \ldots \mathbf{N}] \rightarrow \{0, 1\}$. Given a finite signature of variables, there are only finitely many possible structures with a given value of the variable $\mathbf{N}$.

Our abstract lattice $\mathcal{L}$ is a finite logical language. The concretization function γ yields the extension of a formula, that is, for formula $\phi \in \mathcal{L}$, $\gamma(\phi)$ is the set of structures satisfying ϕ.

The language $\mathcal{L}$ is defined as follows in terms of a set of indexed atomic predicates called R-atoms. We are given a finite set $\mathcal{I}$ of logical symbols to act as indices (say, $\{i, j\}$). For our example structures, an R-atom might be defined as a formula of the form x_k, or $z_k[v]$ or $v = w$, where v and w are scalar variables (y_k) or parameters in $\mathcal{I}$.

For any set of variables $\mathcal{I}$, we will use the shorthand $\dot{\forall} I.Q$ to mean the Q holds for all *distinct* valuations of $\mathcal{I}$ (*i.e.* valuations in which no two distinct variables are equal). Similarly, we will use $\dot{\exists}\mathcal{I}.Q$ to mean that Q holds for some distinct valuation of $\mathcal{I}$. The abstract domain is the set of formulas $\dot{\forall}\mathcal{I}.Q$ where Q, the *matrix*, is a Boolean combination of R-atoms. For example, in our abstract domain we can express the idea that no two processes are in state z_1 by the formula $\dot{\forall} i, j.\ \neg(z_1[i] \wedge z_1[j])$.

We assume that the matrix of our formulas is reduced to a propositional normal form, such as disjunctive normal form. A formula in disjunctive normal form is a disjunction of *minterms*. Each minterm is a conjunction of literals over the R-atoms, in which each R-atom occurs once, either positively or negatively. We can also think of a minterm as a truth assignment to the R-atoms.

The lattice operations are defined by:

$$(\dot{\forall}\mathcal{I}.\ Q_1) \sqcup (\dot{\forall}\mathcal{I}.\ Q_2) = \dot{\forall}\mathcal{I}.\ (Q_1 \vee Q_2) \tag{6}$$

$$(\dot{\forall}\mathcal{I}.\ Q_1) \sqcap (\dot{\forall}\mathcal{I}.\ Q_2) = \dot{\forall}\mathcal{I}.\ (Q_1 \wedge Q_2) \tag{7}$$

That is, least upper bound and greatest lower bound correspond, respectively, to disjunction and conjunction on the matrices.[1] Since conjunction distributes over universal quantification, greatest lower bound is equivalent to conjunction, or intersection in the concrete domain. Thus our abstraction is conjunctive (γ preserves glb's) and we have a unique abstract function α and best transformer $\tau^\sharp$. This abstract domain is also known as an *Indexed Predicate Abstraction* [Lah04] for the given predicate set $\mathcal{R}$.

For any natural number N, the subsystem state space $\mathcal{S}_N$ is defined to be the set of structures in $\mathcal{S}$ such that variable $\mathbf{N}$ is mapped to natural number N. This in turn defines the restricted transformers α_N, γ_N and τ_N. As mentioned above, the invisible invariants method effectively computes the nested fixed point of equation 5 for some chosen fixed value of N. However, it gives up at only one iteration of the middle and outer fixed points. This amounts to computing the concrete reachable states of system N, abstracting them using α_N, and testing whether this is a fixed point of $\tau^\sharp$. Though this approach may seem naïve, in practice it has produced useful inductive invariants in many cases for which manual invariant construction is a non-trivial task [FPPZ06, BPZ06, BPZ07].

3.1 Engineering Invisible Invariants

The invisible invariants approach applies a number of engineering optimizations to make this computation more efficient that might have more general application. These are largely to do with how the various transformers are computed. The method uses symbolic model checking techniques to compute reachable states and exploits symmetries in the system to reduce the computational effort. The key operations are α_N, which is called *project* for reasons that will be clarified shortly, γ_N, which is called *generalize*, and the fixed point test.

Elements of the abstract domain are represented by their matrix. A set $\mathcal{R}_b$ of Boolean variables is used to represent the R-atoms, such that each R-atom p is mapped to a Boolean variable $v_p \in \mathcal{R}_b$. A Binary Decision Diagram (BDD) over these variables can then be used to canonically represent the matrix of the formula.

Assume that $\mathcal{S}_N$ is encoded in some way using Boolean variables $\mathcal{V}_b$ and we have a BDD $S(\mathcal{V}_b)$ representing some set of concrete states $S \subseteq \mathcal{S}_N$. We would like to compute a BDD representing $\alpha_N(S)$. To be precise, we wish to compute the set of minterms over $\mathcal{R}$ that are consistent with S for *some* valuation of the indices, that is:

$$\text{matrix}(\alpha_N(S)) = \vee\{m \in \text{minterms}(\mathcal{R}) \mid \text{for some } s \in S,\ s \models \dot{\exists}\mathcal{I}.\ m\} \tag{8}$$

This set can be computed symbolically using BDD's by defining a relation U_N that defines the truth values of the R-atoms as a function of the concrete state

[1] Note this means that the minterms are the lattice atoms. However, we prefer "minterm" to "atom" here to avoid confusion with the meaning of "atom" in logic.

and the values of the index variables. That is, suppose for a fixed N we encode the values of the index variables in $\mathcal{I}$ with a corresponding set of Boolean variables $\mathcal{I}_b$. For each R-atom a we can construct a Boolean formula $a_N(\mathcal{V}_b, \mathcal{I}_b)$ that is true iff a is true in the concrete state and index valuation encoded by $\mathcal{V}_b$ and $\mathcal{I}_b$ respectively. We then define a relation

$$U_N(\mathcal{R}_b, \mathcal{I}_b, \mathcal{V}_b) = \bigwedge_{a \in \mathcal{R}} (v_a \Leftrightarrow a_N(\mathcal{I}_b, \mathcal{V}_b)). \tag{9}$$

Now we can compute the symbolic representation of $\alpha_N(S)$ as

$$\alpha_N(S)(\mathcal{R}_b) = \dot{\exists} \mathcal{I}_b . \exists \mathcal{V}_b . (S(\mathcal{V}_b) \wedge U_N(\mathcal{R}_b, \mathcal{I}_b, \mathcal{V}_b)) \tag{10}$$

This is just a relational product computation, for which well-established BDD techniques exit. However, we can do better than this by exploiting the symmetry of the state space. That is, because of the restrictions on the logic used to represent the initial state set I and transition relation T, we can guarantee that they are invariant under permutations of the scalar type $[1 \ldots N]$. As a result the set of reachable states also has this symmetry. This means that all distinct valuations of the index variables are equivalent, so if S is invariant under permutation of $[1 \ldots N]$ we have:

$$\alpha_N(S)(\mathcal{R}_b) = \exists \mathcal{V}_b . (S(\mathcal{V}_b) \wedge U_N(\mathcal{R}_b, \mathcal{I}_b, \mathcal{V}_b))(\sigma)$$

where σ is any chosen distinct valuation of the index variables (say $i = 1$, $j = 2$). If there are no scalar variables y_k, this is effectively just *projecting* the state set S onto a fixed set of array elements (say, 1 and 2). Hence, computing the abstraction function α_N symbolically is referred to as projection. Note, however, that we do not require symmetry to compute α_N. It is simply computationally more efficient to substitute fixed values for the index variables than to quantify over them existentially.

Computing γ_N symbolically is just a straightforward interpretation of the language semantics. That is, we compute the set of concrete states satisfying the matrix of the formula, as a function of $\mathcal{I}$, then quantify universally over distinct index valuations:

$$\gamma_N(\phi)(\mathcal{V}_b) = \dot{\forall} \mathcal{I}_b . \exists \mathcal{R}_b . (\phi(\mathcal{R}_b) \wedge U_N(\mathcal{R}_b, \mathcal{I}_b, \mathcal{V}_b)). \tag{11}$$

Finally, we arrive at the fixed point test. Given an element ϕ of the abstract domain, we could of course use a theorem prover to test whether $\tau^\sharp(\phi) \sqsubseteq \phi$. This amounts to checking validity of the formulas $\phi \wedge T \Rightarrow \phi'$ and $I \Rightarrow \phi$. However, we can avoid this call to a theorem prover using a small model theorem. That is, based on the form of these formulas we may be able to show that there is a threshold M such that, if there is a countermodel with $\mathbf{N} > M$, there is a countermodel with $\mathbf{N} \leq M$. Thus it suffices to check $\tau^\sharp_N(\phi) \sqsubseteq \phi$ for all $N \leq M$, which we can do with finite state methods. In particular, this holds when $\tau(\gamma_N(\phi)) \subseteq \gamma_N(\phi)$. In case no small model result is available, we can always fall back on a theorem prover for the fixed point test.

As stated above, the classical Invisible Invariants method computes the reachable states of concrete system N just once, abstracts via α_N, then terminates successfully if this is a fixed point. This corresponds to computing just one iteration of the middle and outer fixed points of equation 5. However, there is no reason in principle to give up at this point if an invariant is not found. One could compute further iterations of $\rho_N^{\sharp *}$. This would mean re-concretizing the abstract state and computing further reachability iterations, repeating this process until a fixed point of $\tau_N^\sharp$ is obtained. If this is not a fixed point of $\tau^\sharp$, one could apply $\tau^\sharp$ (at perhaps a high cost) and continue iterating $\pi_N^\sharp$ to a fixed point of $\tau^\sharp$.

Alternatively, one could apply equation 1 instead of equation 5. In this case the inner iteration computes a fixed point of $\tau_N^\sharp$. At each iteration we go down to the concrete domain via γ_N, one step forward via τ_N, then back up to the abstract domain via α_N. Thus, we effectively alternate concrete steps with project-and-generalize steps (applications of $\gamma_N \circ \alpha_N$). Because we abstract more frequently in this approach, we may reach a fixed point in fewer forward steps, but at an extra cost in computing projections and generalizations.

4 Experiments

We now consider some simple experiments to determine whether the strategy described in Section 2 can effectively reduce the cost of computing precise least fixed points in an abstract domain whose best abstract transformer is expensive to compute. For our abstract domain, we will use Indexed Predicate Abstraction, with a fixed set of predicates $\mathcal{R}$ for each problem.

4.1 Fixed Point Strategies

We will consider three fixed point computation strategies. Strategy A is direct iteration of the best abstract transformer $\tau^\sharp$. Strategy B is the nested strategy of equation 1, which iterates the transformer $\tau_N^\sharp$ in the inner loop. Strategy C is the doubly-nested approach based on equation 5.

As a representative of strategy A, we use the tool Uclid PA [LBC03], which implements Indexed Predicate Abstraction directly, iterating the abstract transformer $\tau^\sharp$. Strategies B and C are implemented using a hybrid of the tools TLV [PS96] and Uclid [BLS02]. The restricted abstraction and concretization operators α_N and γ_N are computed in TLV according to Equations (10) and (11). TLV also computes the concrete transformer τ. For strategies A and B, we use the Uclid tool to compute the abstract transformer, as described below.

4.2 Computing the Abstract Transformer

The Uclid PA tool uses a SAT-based procedure to compute the abstract transformer $\tau^\sharp$. This is based on an eager encoding of the CLU logic into SAT using Uclid, and Boolean quantifier elimination techniques implemented in the tool SATQE.

Though this process is complex, we can think of it naïvely as using counter-models from a theorem prover to compute to a series of under-approximations $\psi_0, \psi_1, \ldots$ to $\tau^\sharp(\phi)$. Suppose we are given a two-vocabulary formula $T(V, V')$ representing the concrete transition relation of the system and that $\phi \sqsupseteq \alpha(I)$ is a pre-fixed point of τ. Our first under-approximation is $\psi_0 = \phi$. Since the ψ_i are under-approximations, we know that $\psi_i = \tau^\sharp(\phi)$ when the following formula is valid:

$$\phi(V) \wedge T(V, V') \Rightarrow \psi_i(V'). \tag{12}$$

We can test this formula with a theorem prover. If it is valid, then we are done. Otherwise, suppose the prover produces a counter-model including an assignment σ' of values to the symbols V' representing the post-state. We know that $\sigma' \in \tau(\gamma(\phi))$, thus $\tau^\sharp(\phi) \sqsupseteq \alpha(\{\sigma'\})$. Our next under-approximation is therefore $\psi_{i+1} = \psi_i \sqcup \alpha(\{\sigma'\})$. We repeat this process until (12) becomes valid (which must eventually occur, since our lattice is finite). We will call this the naïve approach to computing $\tau^\sharp$. Notice that if ϕ is a fixed point to begin with, then only one validity test is required.

The approach of Uclid PA using Boolean quantifier elimination might be significantly more efficient than the naïve approach. Unfortunately, we were not able to run Uclid PA because the source code has been lost. Therefore, for strategy A, we use results previously obtained with Uclid PA by Shuvendu Lahiri on the given set of benchmarks, while for Strategies B and C, we use the naïve method with Uclid as the theorem prover. As it turns out, this is no great disadvantage for strategies B and C, since in all cases, only one call to the theorem prover is needed to confirm an abstract fixed point.

We should note that the actual transformer computed using either method may be weaker than the best transformer, owing to incompleteness of the provers. This is inevitable, as the logics involved are incomplete. There are two particular ways, however, in which Uclid typically fails in practice to recognize a valid formula. First, Uclid Skolemizes the existential quantifiers and eagerly instantiates the universal quantifiers. If the heuristically chosen instantiations are insufficient, a bogus countermodel can result, which can yield an over-approximation of $\tau^\sharp$. Second, a theorem of CLU logic may be needed that must be proved by induction over the natural numbers. It may be necessary to manually augment the formula with such theorems in order to strengthen the abstract transformer. We will see an example of this shortly.

4.3 Benchmarks

For benchmarks, we use three problems solved in [Lah04] using Uclid PA:

1. The "German" cache coherence protocol, with one-message queues.
2. An N-process "Bakery" mutual exclusion algorithm.
3. An N-process version of Peterson's mutual exclusion algorithm from [Din99].

Both mutual exclusion algorithms assume interleaving semantics. The Bakery version assumes that we can find the least non-zero ticket in an atomic step, but uses a loop to wait for each other process in turn. The version of Peterson's algorithm allows testing of the variables of all other processes as an atomic step. The properties being proved are coherence in the first case and mutual exclusion in the other two. The indexed predicate set $\mathcal{R}$ in each case is as given in [Lah04]. Precise descriptions of the problems in the Uclid language can be found at `http://www.kenmcmil.com`.

Table 1 shows the results of computing the least fixed point of $\tau^\sharp$ using the three strategies. The table shows run times for the three strategies, the value of N used and the number of iterations of each transformer required to reach the fixed point (as an example, $2\rho_n^\sharp + 1\tau^\sharp$ would mean that $\rho_n^\sharp$ was computed twice and $\tau^\sharp$ was computed once). Note that the choice of N is a heuristic one. If N is too small, the inner fixed point may not explore enough of the space, resulting in more iterations of the expensive outer fixed point. On the other hand, if N is too large, the BDD representation of the concrete space may explode. Here, we apply a rule of thumb derived from previous work on Invisible Invariants [APR+01], letting N be the number of index variables plus two.

Table 1. Results for computing abstract fixed points with different strategies

	Strategy A		Strategy B		Strategy C		
Problem	Time (s)	Iters	Time (s)	Iters	Time (s)	Iters	N
German	2000	$15\tau^\sharp$	0.08	$8\tau_N^\sharp$	0.60	$1\rho_N^{\sharp*}$	3
Bakery	471	$18\tau^\sharp$	32.00	$16\tau_N^\sharp$	9.80	$1\rho_N^{\sharp*}$	4
Peterson	1000	$18\tau^\sharp$	196.00	$16\tau_N^\sharp$	6.80	$1\rho_N^{\sharp*}$	4

The experiments for strategy A were reported by Lahiri using a 2.1 GHz Pentium machine with 1 GB of RAM. The experiments for strategies B and C were performed by the authors using a single CPU of a 2.4 GHz Intel Core Duo machine. The single CPU performance may vary slightly between these machines. However, we observe that strategy C provides roughly two orders of magnitude improvement in performance. This difference dominates the difference in CPU speeds.

We should note that in Lahiri's work, strategy A is not able to compute a fixed point strong enough to imply mutual exclusion for the Peterson problem. This is because a theorem of CLU logic is needed that cannot be inferred by the prover. We provided this theorem to allow mutual exclusion to be proved.

These experiments validate the intuition behind Invisible Invariants. That is, the best performance is obtained by maximizing the finite-state computations and minimizing the frequency of abstraction operations (though strategy B does score a win for the German protocol). In every case, after computing the concrete reachable states for fixed N and abstracting, we obtain the abstract fixed point and no further iteration is needed. In principle, strategy C can continue to iterate after this point, but in practice this was not needed. The practical difference between these experiments and the Invisible Invariants approach is that we did not require a small model theorem, as the theorem prover was able to perform the abstract fixed point test. In all three cases, only one call to the theorem prover was needed.

5 Related Work

A closely related method is that of Yorsh *et al.* [YBS06]. In that work, rather than systematically exploring a restricted concrete state space as we do here, they explore a random finite subset using a random walk. This subset is then abstracted via α and the abstract fixed-point test is applied. If the test fails, it produces a counter-model. Random exploration then continues from this state. Random walk is also used to under-approximate the abstract transformer in implementations of Van Eijk's method [vE98].

The random walk strategies and our systematic exploration strategy are both in effect using exploration of the concrete state space to under-approximate the abstract transformer. However, they make different practical trade-offs. It may be that a random walk will quickly fill out most or all of the relevant cases in the concrete space and thus provide a close under-approximation of the abstract fixed point. In this case it could be more efficient than a full exploration of a size-bounded model. On the other hand, if significant behavior of the system is reached with low probability in a random walk, then at some point the random exploration may cease to improve the approximation, in which case the number of theorem prover calls could become large. This might be avoided by a more systematic exploration of the state space.

Lacking the necessary tools, we were not able to compare the random and systematic approaches experimentally. It should be noted, however, that it is possible to use a hybrid of random and systematic exploration. The approach of exploration from a single counter-model to the fixed point test can also be used with systematic exploration and might result in fewer theorem prover calls in some cases.

Another closely related approach is taken by Bingham in verifying parameterized cache coherence protocols [Bin08]. In this method the abstraction function α projects the state onto a fixed number $N-1$ of processes. The concrete state space used is $\mathcal{S}_N$. The fixed point strategy is essentially our strategy B. A symmetry argument is used to show that a fixed point of $\tau_N^\sharp$ is necessarily a fixed point of $\tau^\sharp$, thus no computation of $\tau^\sharp$ is actually needed. In the absence of symmetry among processes, however, the full strategy B or C could be applied.

Our operators γ_N and α_N bear a superficial similarity to the the *focus* and *blur* operators of [LAS00]. However, focus is not an under-approximation. Rather, focus and blur are used together to compute an *over-approximation* of the best abstract transformer. There are also numerous works that combined the computation of over-approximate and under-approximate fixed points, such as [GNRT10]. Here we are computing *only* an over-approximate fixed point (so, for example, we cannot falsify safety properties).

6 Conclusion

We have observed that the Invisible Invariants method for verifying parameterized systems can be viewed as a form of Indexed Predicate Abstraction, using a particular fixed point strategy. This strategy relies on having some way of segregating the concrete state space into subsets of increasing size or complexity. The intuition is that at some point the abstract domain will not be able to distinguish between systems of increasing size. Thus we may hope to come close to the true abstract least fixed point by tunneling beneath it, computing a concrete fixed point for a fixed size and then abstracting. Often, we obtain the exact abstract least fixed point in this way. We observed that the Invisible Invariants method can be generalized to obtain a complete generic method for computing abstract least fixed points while minimizing computation of the best abstract transformer.

Experimentally, we observed that the most efficient approach for the parameterized systems we tested is to rely as much as possible on concrete computation and to apply abstraction and concretization as little as possible.

The generic approach is not restricted to systems of finite state processes. It requires only a heuristically useful way of choosing a finite subset of the state space. Thus, for heap manipulating programs, we might restrict to heaps of a bounded sized. The resulting invariant is proved for heaps of unbounded size (see also [BPZ07]). Neither is the method restricted to uniform or symmetric collections of processes. It is possible, for example, to break the symmetry of processes by having the behavior of each process depend on its index. This symmetry breaking only affects the efficiency of computing α_N. Dynamic process creation is also easily modeled.

Restricting the concrete computation to states of a given "size" is one heuristic choice. It could be that other kinds of under-approximations are more effective. For example, we might further restrict the concrete space by adding symmetry-breaking constraints, applying partial-order reductions, and so on. The general fixed-point strategy allows many such possibilities.

The generic strategy should also be applicable in a variety of abstract domains, including, for example, shape analysis domains such as Three-Valued Abstractions [LAS00] and Separation Logic-based domains [BCI10]. We were not able to explore this possibility because of a lack of tools to perform the concrete exploration, so this application remains for future work.

Acknowledgment. Thanks to Francesco Logozzo and the anonymous reviewers for help in improving the presentation of this paper.

References

[APR$^+$01] Arons, T., Pnueli, A., Ruah, S., Xu, J., Zuck, L.D.: Parameterized verification with automatically computed inductive assertions. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, p. 221–234 Springer, Heidelberg (2001)

[BCI10] Berdine, J., Cook, B., Ishtiaq, S.: SLAyer: Memory safety for systems-level code. Technical Report 144848, MSR (2010); To appear in CAV 2011 (2011)

[BFPZ05] Balaban, I., Fang, Y., Pnueli, A., Zuck, L.D.: IIV: An invisible invariant verifier. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 408–412. Springer, Heidelberg (2005)

[Bin08] Bingham, J.D.: Automatic non-interference lemmas for parameterized model checking. In: Cimatti, A., Jones, R.B. (eds.) FMCAD, pp. 1–8. IEEE, Los Alamitos (2008)

[BLS02] Bryant, R.E., Lahiri, S.K., Seshia, S.A.: Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, p. 78. Springer, Heidelberg (2002)

[BPZ06] Balaban, I., Pnueli, A., Zuck, L.D.: Invisible safety of distributed protocols. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4052, pp. 528–539. Springer, Heidelberg (2006)

[BPZ07] Balaban, I., Pnueli, A., Zuck, L.D.: Shape analysis of single-parent heaps. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 91–105. Springer, Heidelberg (2007)

[CC77] Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252 (1977)

[CC79] Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL, pp. 269–282 (1979)

[Cou81] Cousot, P.: Semantic foundations of program analysis. In: Muchnick, S.S., Jones, N.D. (eds.) Program Flow Analysis - Theory and Applications. Prentice Hall software series, pp. 303–342. Prentice Hall, Englewood Cliffs (1981)

[Din99] Dingel, J.: Systematic Parallel Programming. PhD thesis, Carnegie Mellon University (1999)

[FPPZ06] Fang, Y., Piterman, N., Pnueli, A., Zuck, L.D.: Liveness with invisible ranking. STTT 8(3), 261–279 (2006)

[GNRT10] Godefroid, P., Nori, A.V., Rajamani, S.K., Tetali, S.: Compositional may-must program analysis: unleashing the power of alternation. In: POPL, pp. 43–56 (2010)

[Lah04] Lahiri, S.K.: Ubounded System Verification using decision Procedure and predicate abstraction. PhD thesis, Carnegie Mellon University (2004)

[LAS00] Lev-Ami, T., Sagiv, S.: TVLA: A system for implementing static analyses. In: SAS 2000, pp. 280–301. Springer, Heidelberg (2000)

[LBC03] Lahiri, S.K., Bryant, R.E., Cook, B.: A symbolic approach to predicate abstraction. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 141–153. Springer, Heidelberg (2003)

[Mon09] Monniaux, D.: Automatic modular abstractions for linear constraints. In: POPL 2009, pp. 140–151. ACM, New York (2009)

[PS96] Pnueli, A., Shahar, E.: The TLV system and its applications. Technical report, The Weizmann Institute (1996)

[RSY04] Reps, T., Sagiv, M., Yorsh, G.: Symbolic implementation of the best transformer. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 252–266. Springer, Heidelberg (2004)

[vE98] van Eijk, C.A.J.: Sequential equivalence checking without state space traversal. In: Proceedings of the Conference on Design, Automation and Test in Europe, DATE 1998, pp. 618–623. IEEE Computer Society, Washington, DC, USA (1998)

[YBS06] Yorsh, G., Ball, T., Sagiv, M.: Testing, abstraction, theorem proving: better together? In: ISSTA, pp. 145–156 (2006)

[ZP04] Zuck, L.D., Pnueli, A.: Model checking and abstraction to the aid of parameterized systems (a survey). Computer Languages, Systems & Structures 30(3-4), 139–169 (2004)

An Abstraction-Refinement Framework for Trigger Querying

Guy Avni and Orna Kupferman

School of Computer Science and Engineering, Hebrew University, Jerusalem 91904, Israel

Abstract. *Trigger querying* is the problem of finding, given a system M and an LTL formula φ, the set of *scenarios* that trigger φ in M; that is, the language L of finite computations of M such that all infinite computations that have a prefix in L continue with a suffix that satisfies φ. For example, the trigger query $M \models ? \mapsto Ferr$ asks for the set of scenarios after which err aught to eventually happen. Trigger querying thus significantly extends query checking, which seeks propositional solutions, and is an extremely useful methodology for system exploration and understanding. The weakness of trigger querying lies in the fact that the size of the solution is linear in the size of the system. For trigger querying to become feasible in practice, we must offer solutions to cope with systems of big, and possibly infinite, state spaces.

In this paper we describe an abstraction-refinement framework for trigger querying. The general idea is to replace the reasoning about M by reasoning about an abstraction M_A of M, and return to the user two languages, L_l and L_u, that under- and over-approximate L, respectively. We consider predicate abstraction, and the languages L_l and L_u are defined with respect to the set of predicates. The challenge in defining the approximating languages is that trigger querying does not have a clear polarity, and the definition of L_l and L_u has to combine the upper- and over-approximations of M. We describe an automata-theoretic approach for refining and reducing $L_u \setminus L_l$. While refinement for model checking is *lengthwise*, in the sense that it is based on counterexamples, here we suggest both lengthwise and *widthwise* refinement, where the latter is based on cuts in an automaton for $L_u \setminus L_l$ and thus can symbolically handle batches of counterexamples. We show that our framework is robust and can be applied also for classical query checking as well as variants and extensions of trigger querying.

1 Introduction

The field of formal verification developed from the need to verify that a system satisfies its specification. In practice, formal-verification methodologies are used not only for ensuring correctness of systems but also for understanding systems and exploring their models [20]. In [6], Chan suggested to formalize model exploration by means of *query checking*. The input to the query-checking problem is a model M and a query φ, where a query is a temporal-logic formula in which some sub-formula is the place-holder "?". A solution to the query is a propositional assertion that, when replaces the place-holder, results in a formula that is satisfied in M. For example, if the query is $AG(? \rightarrow ack)$, then the set of solutions includes all assertions θ for which $M \models AG(\theta \rightarrow ack)$. A query checker should return the strongest solutions to the query (strongest in the

E. Yahav (Ed.): SAS 2011, LNCS 6887, pp. 263–279, 2011.

sense that they are not implied by other solutions).[1] The work of Chan was followed by further work on query checking, studying its complexity, cases in which only a single strongest solution exists, the case of multiple (possibly related) place-holders, and more [5,8,9,25].

A serious shortcoming of query checking is the fact that the solutions are propositional assertions. Thus, query checking is restricted to questions regarding one point in time, whereas most interesting questions about systems involve scenarios that develop over time. For example, solutions to the query $AG(? \rightarrow ack)$ are propositional assertions that imply ack in all the states of the system. Such assertions are clearly less interesting than scenarios after which ack is valid. As another example, consider a programmer trying to understand the code of some software. In particular, the programmer is interested in situations in which some function is called with some parameter value. The actual state in which the function is called is by far less interesting than the scenario that has lead to it. Query checking does not enable us to reveal such scenarios.

In [21], the authors introduce and study *trigger querying*, which addresses the shortcoming described above. Given a model M and a temporal behavior φ, trigger querying is the problem of finding the set of scenarios that trigger φ in M. That is, scenarios that are feasible in M and for which if a computation of M has a prefix that follows the scenario, then its suffix satisfies φ.[2]

Kupferman and Lustig formalized trigger querying using the temporal operator $\mapsto$ (triggers). The trigger operator was introduced in SUGAR (the precursor of PSL [4], called suffix implication there), and it plays an important role also in popular industrial specification formalisms like ForSpec [1] and System Verilog Assertions (SVA) [26]. Consider a system M with a set P of atomic propositions. A word w over the alphabet 2^P triggers an LTL formula φ in the system M, denoted $M \models w \mapsto \varphi$, if at least one computation of M has w as a prefix, and for every computation π of M, if w is a prefix of π, then the suffix of π from position $|w|$ satisfies φ (note that there is an "overlap" and the $|w|$-th letter of π participates both in the prefix w and in the suffix satisfying φ). The solution to the trigger query $M \models ? \mapsto \varphi$ is then the set of words w that trigger φ in M. Since, as shown in [21], the solution is regular, trigger-querying algorithms return the solution by means of a regular expression or an automaton on finite words. The weakness of trigger querying lies in the fact that the size of the solution is linear in the size of the system. For trigger querying to become feasible in practice, we must offer solutions to cope with systems of big, and possibly infinite, state spaces.

In this paper we describe an *abstraction-refinement framework for trigger querying*. Abstraction is a well known approach for coping with the huge, and possibly infinite, state space of systems [2,13].[3] In particular, in the context of model checking, the counterexample guided abstraction-refinement (CEGAR) method has proven to be very effective [11,12,22]. Recall that the solution to the trigger query $M \models ? \mapsto \varphi$ is a

[1] Note that a query may not only have several solutions, but may also have several strongest solutions.

[2] The definition in [21] is a bit different and allows also vacuous triggers: scenarios that are not feasible in M are considered solutions too.

[3] A different approach to cope with the complexity of trigger querying is studied in [24]. There, the user approximates the trigger by a statistical analysis of traces in the system.

regular language L over the alphabet 2^P. The general idea of our framework is to replace the reasoning about M by reasoning about an abstraction M_A of M, and return to the user two languages, L_l and L_u, that under- and over-approximate L. In more detail, we consider *predicate abstraction*, where the state space of M_A is 2^{Φ}, for a set Φ of propositional assertions on P. The abstraction M_A is a *modal transition system* [23], and has two types of transitions: *may transitions*, which over-approximate the transitions of M, and *must transitions*, which under-approximate them. The approximating languages L_l and L_u are over the alphabet 2^{Φ}, and they satisfy $L_l \subseteq L \subseteq L_u$, with an appropriate adjustment of $\subseteq$ to the different alphabets.

While L_l and L_u under- and over-approximate L, finding them combines both the under- and over-approximations of M. Intuitively, it follows from the fact that the solution to a trigger query does not have a clear polarity: it is not hard to see that $M \models w \mapsto \varphi$ if the set of the states of M that are reachable by tracing w is not empty and all the states in it satisfy $A\varphi$. When we consider an abstraction that under-approximates the transitions of M, two contradicting things happen: (1) we make it harder for words to make it into the solution, as fewer computations can trace w, and (2) we make it easier for words to make it into the solution, as more states satisfy $A\varphi$. Similar and dual difficulties arise when we try to work with an abstraction that over-approximates M or when we search for L_u.[4]

The smaller is the gap between L_l and L_u is, the more informative our approximating trigger languages are. Given a set of predicates Φ, refinement amounts to extending Φ so that $L_u \setminus L_l$ is reduced. We suggest two approaches for refinement. Both approaches are based on a deterministic automaton $\mathcal{C}$ for $L_u \setminus L_l$. As we show, the construction of $\mathcal{C}$ is based on an analysis of the reasons for lack of information, and avoids the complementation of L_l. The first approach, *lengthwise refinement*, is similar to the one performed in CEGAR, and is based on clinging to a single word $\tau \in (2^{\Phi})^*$ accepted by $\mathcal{C}$, and refining both the transitions that $\mathcal{C}$ traverses in its accepting run on τ as well as the accepting state that the run on τ reaches. The second approach, *widthwise refinement*, is possible thanks to the fact $\mathcal{C}$ accepts *all* the words in $L_u \setminus L_l$. Widthwise refinement iteratively reduces the language of $\mathcal{C}$ by clinging to a cut in its underlying graph. As we elaborate in the paper, $\mathcal{C}$ has cuts of special interest – these that correspond to may transitions that are not must transitions in M_A. An advantage of widthwise refinement is that it manipulates sets of states and thus has a simple symbolic implementation. We also suggest a hybrid approach that combines lengthwise and widthwise refinements, by basing the refinement on a sub-language of $L_u \setminus L_l$, and is also symbolic. All the three approaches are complete, in the sense that, unless we start with an infinite-state system, repeated application of them results in $L_l = L = L_u$.

We show that our framework is robust and can handle variants and extensions of trigger querying: classical query checking (in fact, the abstraction-refinement framework there is much simpler), constrained trigger querying (where the input also includes a regular expression, which restricts the range of solutions), and necessary conditions (where the goal is to characterize necessary conditions on the triggers; this problem is only NLOGSPACE-complete in the system).

[4] A different combination of may and must transitions, whose goal is to combine verification and falsification, is suggested also in [17].

Beyond making trigger-querying and its variants feasible in practice, we find the framework interesting from a theoretical point of view as it involves several conceptual differences from CEGAR, and thus involves new general ideas about abstraction and refinement. As we elaborate in Section 6, we believe that these ideas are useful also in other abstraction-refinement methodologies.

Due to the lack of space, some proofs and examples are omitted from this version and can be found in the full version, in the authors' homepages.

2 Preliminaries

We model systems over a set P of atomic propositions by a Kripke structure $M = \langle P, S, S_0, R\rangle$, where $S = 2^P$ is the set of states, $S_0 \subseteq S$ is a set of initial states, and $R \subseteq S \times S$ is a total transition relation. Since we take the set of states to be 2^P, we do specify a labeling function: the set of atomic propositions that hold in state $s \in 2^P$ is simply s. Note that our Kripke structures are deterministic (see Remark 2). A computation of M is a (finite or infinite) sequence of states $\pi = s_1, s_2, \ldots$ such that $s_1 \in S_0$ and $R(s_i, s_{i+1})$ for every $i \geq 1$. For an index $i \geq 1$, we use $\pi[1..i]$ to denote the prefix $s_1, \ldots, s_i$ of π and use π^i to denote the suffix $s_i, s_{i+1}, \ldots$ of π. Note that a word over the alphabet 2^P corresponds to at most one computation in M. The language of M, denoted $L(M)$, is the set of all computations of M.

For a Kripke structure M, a set of states S, and an LTL formula φ, we use $(M, S) \models \varphi$ to indicate that all the computations that start in states in S satisfy φ. When $S = S_0$, we write $M \models \varphi$. Also, when $S = \{s\}$ is a singleton, we write $(M, s) \models \varphi$. We denote by $[\![\varphi]\!]$ the set of states that satisfy φ. I.e., for every $s \in 2^P$ we have that $s \in [\![\varphi]\!]$ iff $(M, s) \models \varphi$.

2.1 Trigger Querying

A word $w \in (2^P)^*$ triggers an LTL formula φ in a Kripke structure M, denoted $w \mapsto \varphi$, if w is a computation of M and for every infinite computation $\pi \in L(M)$, if w is a prefix of π, then the suffix of π from position $|w|$ satisfies φ. Formally, $M \models w \mapsto \varphi$ iff for every computation π of M, if $\pi[1..|w|] = w$ then $\pi^{|w|} \models \varphi$. Note that there is an "overlap" and the $|w|$-th position in π participates in both the prefix w and the suffix satisfying φ. Trigger querying was introduced and studied in [21]. The solution of the trigger query $M \models ? \mapsto \varphi$ is the language of all words triggering φ in M.

Let us consider an example. Assume that M models a hardware design with a signal *err* that is raised whenever an error occurs. We might be interested in characterizing the scenarios after which the signal *err* is raised. These scenarios are the solution to the trigger query $M \models ? \mapsto$ *err*. It may also be the case that we are really interested in characterizing the scenarios after which *err* aught to be raised. The difference is that now we are interested in "crossing the point of no return"; that is, the point from which *err* would eventually (possibly in the distant future) be raised. The set of such scenarios are the solution to the trigger query $M \models ? \mapsto Ferr$.

Remark 1. Our definition is a variant of the one defined in [21]. There, $M \models w \mapsto \varphi$ iff for every infinite computation $\pi \in L(M)$ if $\pi[1 \ldots |w|] = w$ then $\pi^{|w|} \models \varphi$. Thus, all words not in $L(M)$ vacuously trigger all LTL formulas. As discussed in [21], the variants are technically similar. We find the variant with no vacuous solutions more appealing in practice.

The definition of trigger querying refers to computations, rather than states, in M. It is more convenient to work with an equivalent definition, which is state based:

Lemma 1. *[21] For a Kripke structure M, an LTL formula φ, and a finite word $w = s_1, \ldots, s_n$, it holds that $M \models w \mapsto \varphi$ iff $w \in L(M)$ and $s_n \in [\![\varphi]\!]$.*

By Lemma 1, triggering a formula is the same as triggering the set of states that satisfy this formula. Accordingly, we are going to use the notation $M \models w \mapsto T$, for a set $T \subseteq S$. Also, by Lemma 1, the language of triggers is simply the language of M when viewed as an automaton with $[\![\varphi]\!]$ being the set of accepting states. As also follows from Lemma 1, our framework can be extended to additional, more expressive universal formalisms, such as $\forall$CTL*.

2.2 Predicate Abstraction

Consider a concrete Kripke structure $M_C = \langle P, 2^P, S_{0_C}, R_C \rangle$ and a set of predicates $\Phi = \{\theta_1, \ldots, \theta_m\}$ such that θ_i is a Boolean formula over P. Given M_C and Φ, we construct an abstract Kripke structure M_A by merging concrete states that agree on the predicates in φ into a single abstract state. Thus, the set of states of M_A is 2^Φ and a concrete state is mapped to an abstract state if it satisfies exactly all the predicates associated with the abstract state.

For a concrete state $c \in 2^P$ and an abstract state $a \in 2^\Phi$ we say that $c \models a$ iff c satisfies exactly all the predicates in a. Formally, $c \models \bigwedge_{\theta \in a} \theta \ \wedge \ \bigwedge_{\theta \notin a} \neg\theta$. We then also say that $c \in a$. Thus, for convenience, we are going to view an abstract state both as a set of predicates (and use the notation $\theta \in a$, for $\theta \in \Phi$) and as a set of concrete states (and use $c \in a$). Note that the predicates in Φ need not be independent. Thus, some subsets of Φ may be inconsistent, in which case no concrete state corresponds to them.

Typically, Φ is much smaller than P. Consequently, moving to the state space 2^Φ involves loss of information and calls for an approximation of M's transition relation. The abstract structure M_A (also known as a *modal transition system* [23]) has two types of transitions: *may transitions*, which over-approximate these of M, and *must transitions*, which under-approximate them. Formally, $M_A = \langle P, 2^\Phi, S_{0_A}, \rightarrow_{may}, \rightarrow_{must} \rangle$, where S_{0_A}, $\rightarrow_{may}$, and $\rightarrow_{must}$ are defined as follows.

- $a \in S_{0_A}$ iff there exists $c \in a \cap S_{0_C}$,
- $a \rightarrow_{may} a'$ iff there exists $c \in a$ and $c' \in a'$ such that $R_C(c, c')$, and
- $a \rightarrow_{must} a'$ iff for all $c \in a$ there exists $c' \in a'$ with $R_C(c, c')$.

A *may computation* is a sequence of states of M_A in which every two consecutive states have a may transition between them. Formally, $\pi = a_0, a_1, \ldots$ is a may computation if for every $i \geq 0$ it holds that $a_i \rightarrow_{may} a_{i+1}$. A *must computation* is defined in a similar way, with $a_i \rightarrow_{must} a_{i+1}$. Note that every must computation is a may computation, but not vise versa.

Consider an LTL formula φ over Φ and an abstract state a. We distinguish between *may satisfaction* and *must satisfaction*. We say that $(M_A, a) \models_{may} \varphi$ if for every infinite may computation π that starts in a, we have $\pi \models \varphi$. Must satisfaction is defined similarly. Since may computations over-approximate the set of concrete computations, and must computations under-approximate them, and since the more computations there are, the less likely it is to satisfy an LTL formula, we have the following.

Lemma 2. *[16] Consider an LTL formula φ over Φ.*

1. *If $(M_A, a) \models_{may} \varphi$ then for every $c \in a$ it holds that $(M_C, c) \models \varphi$.*
2. *If $(M_A, a) \not\models_{must} \varphi$ then for every $c \in a$ it holds that $(M_C, c) \not\models \varphi$.*

For a concrete Kripke structure M_C and a set Φ of predicates, we denote by $M_C(\Phi)$ the abstract Kripke structure with state space 2^{Φ} that is induced by M_C.

Remark 2. Recall that our Kripke structures are deterministic. It is possible to define trigger querying also for nondeterministic systems – this is also the setting in [21], which make the trigger-querying problem PSPACE-complete in the size of the system. As in LTL model checking, abstraction is essential also when the complexity is only NLOGSPACE in the system. We will discuss the nondeterministic setting further in Remark 3.

2.3 Relating Concrete and Abstract Languages

We relate words over predicates with words over atomic propositions. We define two functions: $abs : 2^P \to 2^{\Phi}$ and $conc : 2^{\Phi} \to 2^{2^P}$. For $c \in 2^P$, we define $abs(c) = \{\theta \in \Phi : c \models \theta\}$. For $a \in 2^{\Phi}$ we define $conc(a) = \{c \in 2^P : c \models a\}$. Thus, $abs(c)$ is the abstract state to which c belongs, and $conc(a)$ is the set of concrete states that belong to a.

We extend the definition of $conc$ and abs to words. For a finite or infinite word $w = w_1, w_2, \ldots$ over 2^P we define $abs(w)$ to be the word $\tau = \tau_1, \tau_2, \ldots$ over 2^{Φ} of the same length as w such that for every $i \geq 1$ it holds that $abs(w_i) = \tau_i$. For a word $\tau = \tau_1, \tau_2, \ldots$ over 2^{Φ} we define the language $conc(\tau)$ as the set of words w such that $\tau = abs(w)$. That is, for every $w = w_1, w_2, \ldots \in conc(\tau)$ and $i \geq 1$ it holds that $w_i \in conc(\tau_i)$.

For an abstract structure $M_C(\Phi)$ and an abstract may or must computation $\tau = \tau_1, \tau_2, ..$ of M_A, we say that τ has a *matching* concrete computation iff there is a computation $w \in conc(\tau) \cap L(M_C)$. Note that if τ is a must computation then it always has a matching concrete computation, but if τ is a may computation, it need not have a matching concrete computation.

We relate languages over predicates with languages over atomic propositions. For languages $L \subseteq (2^P)^*$ and $T \subseteq (2^{\Phi})^*$ we say that $L \subseteq T$ iff for every $w \in L$ it holds that $abs(w) \in T$. Defining language containment in the other direction is more involved, as $conc(\tau)$, for $\tau \in (2^{\Phi})^*$, contains many concrete computations and not all of them may be of interest. Therefore, in addition to the usual $T \subseteq L$ relation, we define a variant of containment that has a concrete Kripke structure M_C as an additional parameter. For a single word $\tau \in (2^{\Phi})^*$ and a language $L \subseteq (2^P)^*$, we say that τ *is* *in L with respect to M_C*, denoted $\tau \in_{M_C} L$, if $conc(\tau) \cap L(M_C) \subseteq L$ and $conc(\tau) \cap L(M_C) \neq \emptyset$. That is, for every concrete word $w \in conc(\tau)$, if $w \in L(M_C)$ then w in L, and there is a word $w \in conc(\tau)$ that satisfies this condition non-vacuously. Now, for languages $T \subseteq (2^{\Phi})^*$ and $L \subseteq (2^P)^*$, we say that $T \subseteq_{M_C} L$ if for every $\tau \in T$, it holds that $\tau \in_{M_C} L$.

3 Approximating Triggers

Let M_C be a concrete Kripke structure, Φ a set of predicates, and φ an LTL formula over Φ. Also, let $M_A = M_C(\Phi)$ and $L_c \subseteq (2^P)^*$ be the solution to the trigger query $M_C \models ? \mapsto \varphi$. That is, for a word w over 2^P, it holds that $w \in L_c$ iff $M_C \models w \mapsto \varphi$. As discussed above, a nondeterministic automaton for L_c is exponential in the size of M_C, and our goal is to replace it by approximating languages by reasoning about M_A. Thus, given M_C, Φ, and φ, our goal is to find two languages $L_l, L_u \subseteq (2^{\Phi})^*$ such that $L_l \subseteq_{M_C} L_c \subseteq L_u$.

We distinguish between *may-triggering* and *must-triggering*. Consider an abstract Kripke structure M_A. For a word $\tau = a_1, \ldots, a_n$ over 2^{Φ} and a set of states $S \subseteq 2^{\Phi}$ we say that $\tau \mapsto_{may} S$ iff τ is a may computation of M_A and $a_n \in S$. Similarly, $\tau \mapsto_{must} S$ iff τ is a must computation of M_A and $a_n \in S$.

We use $M_A \models \tau \mapsto_{\alpha} [\![\varphi]\!]_{\beta}$, where $\alpha, \beta \in \{may, must\}$, to denote that τ α- triggers the set of states that β-satisfy φ.

We define the two languages $L_l, L_u \subseteq (2^{\Phi})^*$ as follows:

- $L_l = \{\tau \in (2^{\Phi})^* : M_A \models \tau \mapsto_{must} [\![\varphi]\!]_{may}\}$.
- $L_u = \{\tau \in (2^{\Phi})^* : M_A \models \tau \mapsto_{may} [\![\varphi]\!]_{must}\}$.

Note that L_l and L_u are defined by M_A when viewed as a deterministic automaton. For L_l, the automaton follows the must transitions and its accepting states are $[\![\varphi]\!]_{may}$. For L_u, it follows may transitions and its accepting states are $[\![\varphi]\!]_{must}$. Thus, the complexity is still linear in the system, but now it is the abstract system, which is considerably smaller.

Recall that $L_c = \{w \in (2^P)^* : M_C \models w \mapsto [\![\varphi]\!]\}$. Intuitively, L_l under-approximates L_c as words τ in L_l should pass two criteria that are more difficult to pass than these that words in L_c should: First, the word has to be a must (rather than concrete) computation. Second, the last state in the path should may satisfy (rather than satisfy) φ. Likewise, L_u over-approximates L_c as words τ in L_u should pass two criteria that are less difficult to pass than these that words in L_c should: the word has to be a may computation and the last state in the path should must satisfy φ. We now prove this intuition formally.

Theorem 1. $L_l \subseteq_{M_C} L_c$.

Proof: Let $\tau = a_1, \ldots, a_n \in L_l$. We show that $\tau \in_{M_C} L_c$. Thus, $L(M_C) \cap conc(\tau)$ is not empty and for every $w \in L(M_C) \cap conc(\tau)$ it holds that $M_C \models w \mapsto \varphi$. We first prove that $L(M_C) \cap conc(\tau)$ is not empty. Recall that τ is a must computation and so there must be a valid concrete computation $w = c_1, \ldots, c_n$ such that for all $1 \leq i < n$ it holds that $c_i \in a_i$. Clearly, $w \in conc(\tau)$ and $w \in L(M_C)$.

Next we prove that for every $w \in L(M_C) \cap conc(\tau)$ it holds that $M_C \models w \mapsto \varphi$. Consider a word $w = c_1, \ldots, c_n \in L(M_C) \cap conc(\tau)$. We show that $(M_C, c_n) \models \varphi$. Since $\tau \in L_l$, we know that τ is a must computation in M_A. By definition, for $1 \leq i \leq n$ it holds that $c_i \in a_i$. By Lemma 2, $a_n \in [\![\varphi]\!]_{may}$ implies that for every $c \in a_n$ it holds that $(M_C, c) \models \varphi$, in particular $(M_C, c_n) \models \varphi$, and we are done. □

Theorem 2. $L_c \subseteq L_u$.

Proof: Consider a word $w \in L_c$. We prove that $abs(w) \in L_u$. That is, for the word $\tau = abs(w)$, it holds that $\tau \mapsto_{may} [\![\varphi]\!]_{must}$. By definition, $w \in L_c$ implies that $w \in L(M_C)$. Let $w = c_1, \ldots, c_n$. Consider the sequence $\tau = a_1, \ldots, a_n$ of M_A, where for every $1 \leq i \leq |w|$ it holds that $a_i = abs(c_i)$. Since for every $1 \leq i < n$ it holds that $R_C(c_i, c_{i+1})$, then $a_i \rightarrow_{may} a_{i+1}$. By definition, $w \in L_c$ also implies that $c_n \models \varphi$. By Lemma 2, this implies that $a_n \in [\![\varphi]\!]_{must}$. We conclude that $\tau \mapsto_{may} [\![\varphi]\!]_{must}$, and we are done. □

For a word $\tau \in (2^{\Phi})^*$, we say that Φ is *informative* for τ if $\tau \notin L_u$ or $\tau \in L_l$. Thus, refining M_A with respect to Φ is sufficient in order to know whether the words in $conc(\tau) \cap L(M_C)$ trigger φ in M_C: either $\tau \notin L_u$, in which case they all do not, or $\tau \in L_l$, in which case they do. In Example 1 and 2 we show words that are uninformative.

Remark 3. The proofs of Theorems 1 and 2 depend on M_C being deterministic. As we demonstrate in the full version, the theorems are not valid in the nondeterministic setting. In the full version we also suggest an alternative definition for L_l and L_u, with which the theorems are valid. Since the added technical difficulties are orthogonal to the ones addressed in this paper, we prefer to focus on the deterministic setting. We still describe below the alternative definitions. First, for the lower bound, we define L_l^{nd} to be $\{\tau \in (2^{\Phi})^* : \tau \mapsto_{may} [\![\varphi]\!]_{may}\} \cap L_{must}(M_A)$. Informally, τ is in L_l^{nd} iff every may computation that induces τ ends in a state that may-satisfies φ, and there is a must computation that induces τ. Now, for the upper bound, we define L_u^{nd} to be all the words $\tau \in (2^{\Phi})^*$ such that there is a may computation $a_1, ..., a_n$ that induces τ and $(M_A, a_n) \models_{must} \varphi$. Note that when applied to deterministic systems, we have that L_l^{nd} and L_u^{nd} coincide with L_l and L_u, respectively.

4 Refinement

The search for approximated triggers starts with a set of predicates Φ and two languages L_l and L_u. During the refinement process we iteratively close the gap between L_l and L_u by adding predicates to Φ. In this section we analyze the reasons why Φ need not be informative with respect to some words, and describe an automata-theoretic approach for characterizing the non-informative words and refinement.

4.1 Between the over and under Approximations

We start with examples explaining four different types of "misses" in the approximating languages.

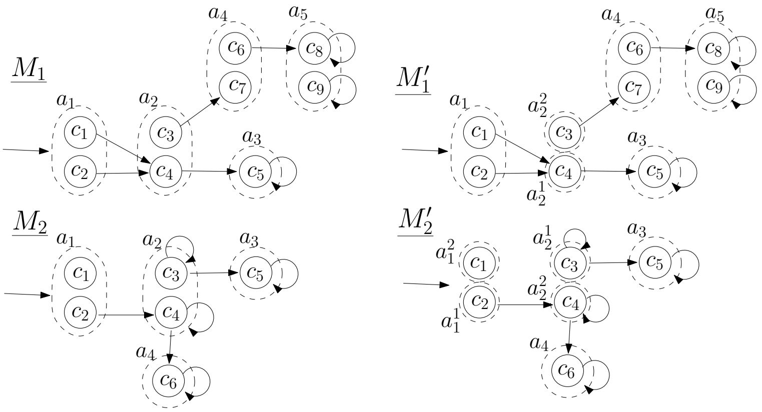


Fig. 1. Information loss in L_u and L_l

Example 1. In this example we demonstrate the case where the word τ triggers φ in M_C but is not in the under-approximation. Formally, $\tau \in_{M_C} L_c$ and $\tau \notin L_l$.

Consider the Kripke structure M_1 appearing in Figure 1. Let M_1^A be its abstraction with state space $\{a_1, \ldots, a_5\}$. Consider the formula $\varphi_1 = FGa_3 \wedge G\neg a_5$ and the word $\tau_1 = a_1 a_2 a_3$. Note that $conc(\tau_1) \cap L(M_C) = \{c_1 c_4 c_5, c_2 c_4 c_5\}$, and that both computations trigger φ_1. Indeed, they both end in c_5 and the only computation that starts in c_5 satisfies φ_1. Hence, $\tau_1 \in_{M_1} L_c$. On the other hand, $\tau_1 \notin L_l$, as τ_1 is a may computation that is not a must computation in M_1^A.

Consider now the word $\tau_2 = a_1 a_2$. Note that $conc(\tau_2) \cap L(M_1) = \{c_1 c_4, c_2 c_4\}$, and that both computations trigger φ_1. Indeed, they both end in c_4 and the only computation that starts in c_4 satisfies φ_1. Hence, $\tau_2 \in_{M_1} L_c$. On the other hand, $\tau_2 \notin L_l$. While it is a must computation in M_1^A, it ends in the state a_2, which does not may satisfy φ_1.

Example 2. In this example we demonstrate the case where the word τ does not trigger φ in M_C but is in the over-approximation. Formally, $\tau \in L_u$ and $\tau \notin_{M_C} L_c$.

Consider the Kripke structure M_2 appearing in Figure 1. Let M_2^A be its abstraction with state space $\{a_1, \ldots, a_4\}$. Consider the formula $\varphi_2 = Ga_2 \vee Ga_3$ and the word $\tau_3 = a_1 a_2 a_3$. Since τ_3 is a may computation in M_2^A that ends in a_3, which must satisfies Ga_3, we have that $\tau_3 \in L_u$. Nevertheless, there is no concrete computation that matches τ_3, so $\tau_3 \notin_{M_2} L_c$.

Consider now the word $\tau_4 = a_1 a_2$. Again, since τ_4 is a may computation and a_2 must satisfies φ_2, we have that $\tau_4 \in L_u$. Note that $(M_2, c_4) \not\models \varphi_2$ because the concrete computation $c_4 c_6^\omega$ does not satisfy φ_2. Since $c_2 c_4 \in conc(\tau_4) \cap L(M_2)$ we have $\tau_4 \notin_{M_2} L_c$.

Note that, for technical convenience, in both examples we use $c_1, c_2, \ldots$ and $a_1, a_2, \ldots$. Nevertheless, the structures can be generated using "real" propositions and predicates. For example, consider the following assignment of propositions to the variables in M_2. Let $P = \{a, b, c, d\}$, $c_1 = \{a, c\}$, $c_2 = \{a, c, d\}$, $c_3 = \{a, b\}$, $c_4 = \{a, b, d\}$, $c_5 = \{a, d\}$, and $c_6 = \{d\}$. By setting $\Phi = \{d \wedge \neg a, a \wedge c, a \wedge b\}$ we get: $a_1 = \{a \wedge c\}$, $a_2 = \{a \wedge b\}$, $a_3 = \emptyset$, and $a_4 = \{d \wedge \neg a\}$.

Our refinement procedure is based on a deterministic automaton $\mathcal{C}$ over the alphabet 2^{Φ} that accepts exactly all the words with respect to which Φ is not informative. In other words, $L(\mathcal{C}) = L_u \setminus L_l$. Rather than constructing $\mathcal{C}$ by taking the product of the automata for L_u and $\overline{L_l}$, we construct it according to an analysis of words with respect to which Φ is not informative. While the examples above demonstrate four possibilities for a word $\tau = a_1, \ldots, a_n \in (2^{\Phi})^*$ to be in $L_u \setminus L_l$, we shall prove that we can group them into two types: Either τ is a may computation that is not a must computation in M_A and $a_n \models_{must} \varphi$, or τ is a must computation in M_A and $a_n \models_{must} \varphi$ but $a_n \not\models_{may} \varphi$.

Accordingly, $\mathcal{C}$ maintains two copies of M_A. In the first copy, $\mathcal{C}$ follows the must transitions of M_A, and it accepts words that end in a state that must satisfies but does not may satisfy φ. The automaton $\mathcal{C}$ moves from the first copy to the second one when it follows a may transition that is not a must transition. In the second copy, $\mathcal{C}$ follows may transitions, and it accepts words that end in a state that must satisfies φ. Formally, $\mathcal{C} = \langle 2^{\Phi}, (2^{\Phi} \times \{1, 2\}) \cup \{a_{init}\}, \delta_C, \{a_{init}\}, F_C \rangle$, where δ_C and F_C are defined as follows (when the condition does not hold, there is no transition and the run gets stuck):

- $\delta_C(a_{init}, a') = \langle a', 1\rangle$, if $a' \in S_{0_A}$.
- $\delta_C(\langle a, 1\rangle, a') = \langle a', 1\rangle$, if $a \rightarrow_{must} a'$. Note that this implies that $a \rightarrow_{may} a'$ too.
- $\delta_C(\langle a, 1\rangle, a') = \langle a', 2\rangle$, if $a \rightarrow_{may} a'$ and $a \not\rightarrow_{must} a'$.
- $\delta_C(\langle a, 2\rangle, a') = \langle a', 2\rangle$, if $a \rightarrow_{may} a'$.
- $F_C = ((\llbracket\varphi\rrbracket_{must} \setminus \llbracket\varphi\rrbracket_{may}) \times \{1\}) \cup (\llbracket\varphi\rrbracket_{must} \times \{2\})$.

Lemma 3. $L(\mathcal{C}) = L_u \setminus L_l$.

4.2 The Refinement Algorithm

Before we turn to describe how we use $\mathcal{C}$ in the process of refinement, let us review the classical counterexample guided abstract refinement (CEGAR) methodology for verification of LTL properties (see [11]). The methodology is based on the fact that if an abstraction that over-approximates the concrete structure M_C satisfies an LTL formula, then so does M_C, and if the abstraction does not satisfy the LTL formula, then a counterexample for the satisfaction can be used for refining the abstraction. Formally, in CEGAR we model check M_A with may transition only. If $M_A \models \varphi$, then we are guaranteed that $M_C \models \varphi$, and we are done. If $M_A \not\models \varphi$, then we get a computation π in $L(M_A)$ such that $\pi \not\models \varphi$ and check whether π corresponds to a concrete computation. If it does, we conclude that $M_C \not\models \varphi$ and we are done. Otherwise, the abstract computation π is spurious and we use it in order to refine M_A to a new abstract structure M'_A that no longer has π as a computation. In the case of predicate abstraction, the refinement is done by adding predicates.

Consider the over and under approximations L_u and L_l of L_c. Let us use the notations $L_u(\Phi)$, $L_l(\Phi)$, and $\mathcal{C}(\Phi)$ in order to indicate that L_u, L_l, and $\mathcal{C}$ have been defined with respect to the set Φ of predicates. The objective of the refinement algorithms is to tighten the gap between L_u and L_l. That is, we start with an initial set of predicates Φ and we refine the set to Φ' so that $L_u(\Phi') \setminus L_l(\Phi')$ is smaller than (in fact, strictly contained in) $L_u(\Phi) \setminus L_l(\Phi)$. In the case of CEGAR, refinement is *lengthwise*, in the sense that it is based on one counterexample that forms a path in the graph of M_A. In our case, we introduce, in addition to lengthwise refinement, also *widthwise* refinement. This is possible thanks to the automaton $\mathcal{C}$, which maintains all the words in $L_u \setminus L_l$, and thus constitutes a compact presentation of all "counterexamples". We also suggest a hybrid approach that combines lengthwise and widthwise refinements. Below we describe the three approaches in detail.

Describing the approaches, we use the *split* operator, defined below. Consider two sets of predicates Φ and Φ' such that $\Phi \subseteq \Phi'$. That is, Φ' extends Φ. For a state $a \in 2^{\Phi}$ we denote by $split(a, \Phi')$ the refinement of a with respect to Φ'. Formally $split(a, \Phi') = \{a' \in 2^{\Phi'} : a' \cap \Phi = a\}$. Thus, all the sets in $split(a, \Phi')$ agree with a on the predicates in Φ and differ on the predicates in $\Phi' \setminus \Phi$. We extend the definition of the split operator to words. For $\tau = a_1, \ldots, a_n \in (2^{\Phi})^*$ we define $split(\tau, \Phi') = \{a'_1, \ldots, a'_n \in (2^{\Phi'})^n : a'_i \in split(a_i, \Phi)$ for all $0 \leq i \leq n\}$.

Lengthwise refinement. The lengthwise refinement procedure, `refineWord`, gets as input a concrete Kripke structure M_C, a set of predicates Φ, and a word $\tau \in (2^{\Phi})^*$ such that Φ is not informative with respect to τ. It then refines $M_C(\Phi)$ according to τ. Thus, the output is a set $\Phi' \supset \Phi$ such that Φ' is informative with respect to all computations $\tau' \in split(\tau, \Phi')$. We note that the procedure can get as input also a concrete word $w \in (2^P)^*$. It then executes `refineWord` with respect to $abs(w)$.

Consider a word $\tau \in L_u \setminus L_l$. Thus, $\tau \in L(C)$. The procedure `refineWord` proceeds in two steps. In the first step, we extend Φ to $\hat{\Phi}$ such that no computation in $split(\tau, \hat{\Phi})$ gets to the second copy of $\mathcal{C}$. In the second step we extend $\hat{\Phi}$ to Φ' so that the accepting states in the first copy of $\mathcal{C}$ do not include states that are reachable by computations in $split(\tau, \Phi')$.

For the first step, we initialize $\hat{\Phi}$ to Φ and extend it iteratively as follows. Let a_{init}, $\langle a_1, b_1 \rangle$, ..., $\langle a_n, b_n \rangle$ be the accepting run of $\mathcal{C}(\Phi)$ on τ. If $b_n = 2$, then τ is a may computation that is not a must computation. We then find the first index $1 \leq i < n$ for which $b_{i+1} = 2$. Note that $a_i \rightarrow_{may} a_{i+1}$ but $a_i \not\rightarrow_{must} a_{i+1}$. We add to $\hat{\Phi}$ a predicate ρ that splits the abstract state a_i into two abstract states: a_i^1 consists of the concrete states that have outgoing edges into concrete states in a_{i+1}, and a_i^2 consists of the states that do not have outgoing edges into the concrete states in a_{i+1}. Thus, after refining, $a_i^1 \rightarrow_{must} a_{i+1}$ and $a_i^2 \not\rightarrow_{may} a_{i+1}$. Note that taking $\rho = \bigvee_{c \in a_i :\, \exists c' \in a_{i+1} \text{ with } R_C(c,c')} c$ achieves this goal. We continue with this step as long as there is a word in $split(\tau, \hat{\Phi})$ that $\mathcal{C}(\hat{\Phi})$ accepts with a run that ends in the second copy.

We start the second step with $\hat{\Phi}$, so the runs of $\mathcal{C}(\hat{\Phi})$ on all words in $split(\tau, \hat{\Phi})$ end in the first copy. Recall that the set of accepting states in this copy is $([\![\varphi]\!]_{must} \setminus [\![\varphi]\!]_{may}) \times \{1\}$. In order to remove a state $\langle a, 1 \rangle$ from F_C we use standard CEGAR, which studies counterexamples to the may-satisfaction of φ in a. As in CEGAR, if the counterexample is spurious, we use it to refine M_A so that may-satisfaction is challenged. Unlike standard CEGAR, here the procedure does not terminate when we detect a counterexample that is not spurious. Instead, such a counterexample witnesses that the must-satisfaction of φ in a is due to under-approximation, and we use it in order to refine M_A so that must-satisfaction is challenged.

Example 3. As a first example, consider the Kripke structure M_1 in Figure 1, its abstraction M_1^A, the formula $\varphi_1 = FGa_3 \wedge G\neg a_5$, and the computation $\tau_1 = a_1 a_2$. As shown in Example 1, $\tau_1 \in L_u \setminus L_l$. Since τ_1 is a must computation, the accepting run of $\mathcal{C}$ on τ_1 ends in the state $\langle a_2, 1\rangle$. Accordingly, we do not perform iterations in the first step of `refineWord` and continue to the second step, where CEGAR methods return the computation $\pi_1 = a_2 a_4 a_5^\omega$. We then find the state a_2 as a failure state and return the predicate $\rho_1 = c_3$ (note that only c_3 has an edge to states in a_4). We split the state a_2 (see M_1' on the right side of Figure 1). Note that all the computations starting at a_2^1 are concrete computations. It follows that a_2^1 is no longer an accepting state and we terminate the procedure. Note also that after the refinement, the word $a_1 a_2^1$, which is the only word that is both in $\mathit{split}(\tau_1, \{a_1, a_2^1, a_2^2, a_3, a_4, a_5\})$ and a computation in the final abstract structure, is in L_l as required.

As a second example, consider the Kripke structure M_2 in Figure 1, its abstraction M_2^A, the formula $\varphi_2 = Ga_2 \vee Ga_3$, and the word $\tau_2 = a_1 a_2 a_3$. As shown in Example 2, $\tau_2 \in L_u \setminus L_l$. Since τ_2 is a may computation that is not a must computation, the accepting run of $\mathcal{C}$ on τ_2 ends in the state $\langle a_3, 2\rangle$. We perform an iteration of the first step of `refineWord`. The failure state is a_1 and we add the predicate c_2, which splits a_1 into a_1^1 and a_1^2. We continue to another iteration of the first step and find the word $a_1^1 a_2 a_3$. The run on it ends in the state $\langle a_3, 2\rangle$. The failure state is a_2 and we split it by adding the predicate c_3. We construct the abstract structure $M_A(\{a_1^1, a_1^2, a_2^1, a_2^2, a_3, a_4\})$ (see M_2' on the right side of Figure 1). Since all the edges are now concrete edges, $L_l = L_u$, and we skip the second step of `refineWord`.

Widthwise refinement. For two sets of abstract states $S, T \subseteq 2^\Phi$, we say that the pair $\langle S, T\rangle$ induces an *interesting frontier* in $\mathcal{C}$ if (1) all the states in $S \times \{1\}$ are reachable in $\mathcal{C}$ from s_{init}, and (2) all the states in $T \times \{2\}$ can reach an accepting state in $\mathcal{C}$. Interesting frontiers are interesting indeed: if there are two states $a \in S$ and $a' \in T$ such that $a \rightarrow_{may} a'$ but $a \not\rightarrow_{must} a'$, then the transition from $\langle a, 1\rangle$ to $\langle a', 2\rangle$ participates in an accepting run of $\mathcal{C}$. We refer to a pair $\langle a, a'\rangle$ as above as a *bridge* in $\langle S, T\rangle$.

Widthwise refinement is based on a calculation of interesting frontiers and elimination of their bridges. The refinement procedure `refineCut` calculates frontiers that are not only interesting but also constitute a cut in the graph of $\mathcal{C}$: every accepting run of $\mathcal{C}$ that ends in the second copy must contain a bridge. Thus, as $\mathcal{C}$ is deterministic, elimination of bridges necessarily reduces the language of $\mathcal{C}$.

Consider a set of abstract states $P \subseteq 2^\Phi$. We define $\mathit{post}^1_{\mathcal{C}}(P)$ as the set of states in the first copy of $\mathcal{C}$ that have incoming edges from states in P. Formally, $\mathit{post}^1_P(S) = \{a' :$ there exists $a \in P$ such that $\langle a', 1\rangle \in \delta_C(\langle a, 1\rangle, a')\}$. We define $\mathit{pre}^2_{\mathcal{C}}(P)$ as the set of states in the second copy of $\mathcal{C}$ that have outgoing edges into states in P. Formally, $\mathit{pre}^2_{\mathcal{C}}(P) = \{a :$ there exists $a' \in P$ such that $\langle a', 2\rangle \in \delta_C(\langle a, 2\rangle, a'\}$. The procedure `refineCut`, described in Figure 2, starts with the interesting frontier $\langle S_{0A}, [\![\varphi]\!]_{must}\rangle$ (note that indeed, all states in $S_{0A} \times \{1\}$ are reachable from the initial state of $\mathcal{C}$, and all the states in $[\![\varphi]\!]_{must} \times \{2\}$ are accepting in $\mathcal{C}$), and iteratively apply $\mathit{post}^1_{\mathcal{C}}$ and $\mathit{pre}^2_{\mathcal{C}}$ on the sets found reachable so far. The sets can be maintained by BDDs and their update is symbolic. The termination of `refineCut` is determined by the user. In the description below we guard the iterations by a Boolean flag *cont* that the user may update in the **update**(cont) procedure. Several updates are possible:

```
Procedure refineCut;
Input: a set of predicates $\Phi$
Output: a set of predicates $\Phi'$
$\Phi' \leftarrow \Phi$ ; $S \leftarrow S_{0A}$ ; $T \leftarrow [\![\varphi]\!]_{must}$ ;
while $cont$ do
    $S \leftarrow post^1_{\mathcal{C}}(S) \cup S$;
    $T \leftarrow pre^2_{\mathcal{C}}(T) \cup T$;
    update$(cont)$;
end
$B \leftarrow (S \times T) \cap (\rightarrow_{may} \setminus \rightarrow_{must})$ ;
$\Phi' \leftarrow$refine$(B, \Phi)$;
```

Fig. 2. The symbolic refineCut procedure

the procedure can run for a bounded number of iterations (which may be a parameter to the procedure), until a fixed-point is reached (which guarantees that $\mathcal{C}(\Phi')$ accepts only must computations, and is therefore typically too good), or until a desired number of bridges is accumulated. The procedure also uses the procedure **refine**, which, as described above, splits the states in the sources of bridges so that they are no longer bridges.

Note that refineCut is similar to step one of refineWord in that it only refines paths that correspond to words in $L(\mathcal{C})$. It does not refine the accepting states like step two of refineWord (that is, such states may be refined as a result of moving to Φ', but they do not play a role in deciding which predicates to add).

Hybrid refinement Recall that lengthwise refinement clings to the transitions $\mathcal{C}$ traverses when a single word is read. Dually, widthwise refinement clings to a cut in $\mathcal{C}$ that contains a single transition in a run of many accepted words. Hybrid refinement combines the two approaches by clinging to a *language* of words.

Hybrid refinement gets from the user a regular expression r over 2^{Φ} of words he wants the approximating languages to be informative about. As with lengthwise refinement, the input can also be given as an expression over 2^P, in which case we replace $c \in 2^P$ by $abs(c)$. The procedure refineLanguage then constructs a nondeterministic automaton $\mathcal{A}$ for $L(r)$ and runs refineCut on the product of $\mathcal{C}$ with $\mathcal{A}$. Accordingly, the frontier and bridges are limited to words accepted by both $\mathcal{C}$ and $\mathcal{A}$.

5 Variants of Trigger Querying

In this section we consider several variants of trigger querying and show that our framework is robust and can handle them too. We start with the classical (non-triggered) query-checking problem, where an abstraction-refinement framework is quite straightforward.

5.1 Query Checking

The input to the LTL *query-checking* problem is a model M over a set P of atomic propositions and a query φ, where a query is an LTL formula in which some subformula is the place-holder ? (e.g., $AG?$). The solution to the query-checking problem, denoted $QC(M, \varphi)$, is the set of strongest propositional assertions over P that, when replace the place-holder, result in a formula that is satisfied by M. We use $\varphi[? \leftarrow \theta]$ to denote that formula obtained from φ by replacing ? by θ. So, $\theta \in QC(M, \varphi)$ iff $M \models \varphi[? \leftarrow \theta]$ and for all propositional assertions ξ over P, if $\xi \rightarrow \theta$ then $M \not\models \varphi[? \leftarrow \xi]$.

Note that we consider here queries in LTL. Adjusting the framework to branching temporal logic is possible; it is more complicated, as it combines may and must transitions, but the expected thing works, and we leave it out of the scope of our contribution. In particular, for branching temporal logic, researchers have already found methods to cope with the complexity of query checking [5,19]. On the other hand, known algorithms for solving LTL query checking do not do much better than checking all possible solutions. A nice exception, based on integer linear programming, is presented in [10], but it works only on a subclass of queries.

Abstraction for query checking For two sets of propositional assertions Γ_1 and Γ_2, we say that $\Gamma_1 \tilde{\subseteq} \Gamma_2$ if for every $\theta \in \Gamma_1$, there exists $\xi \in \Gamma_2$ such that $\xi \rightarrow \theta$ (possibly $\xi = \theta$). Thus, all the propositional formulas in Γ_1 are implied by these in Γ_2. In the *abstract query-checking* problem, we are given a concrete Kripke structure M_C, a set of predicates Φ, and an LTL query φ over Φ. The goal is to find two sets, Γ_l and Γ_u, of propositional assertions over Φ that under- and over-approximate the set of solutions. Formally, $\Gamma_l \tilde{\subseteq} QC(M_C, \varphi) \tilde{\subseteq} \Gamma_u$.

As we show below, the straightforward thing to do, namely to reason about the over- and under-approximations of M_C, work. Formally, let $M_A^{may} = \langle \Phi, 2^{\Phi}, S_{0_A}, \rightarrow_{may} \rangle$ and $M_A^{must} = \langle \Phi, 2^{\Phi}, S_{0_A}, \rightarrow_{must} \rangle$. Then,

Theorem 3. $QC(M_A^{may}, \varphi) \tilde{\subseteq} QC(M_C, \varphi) \tilde{\subseteq} QC(M_A^{must}, \varphi)$.

Refinement for query checking Let $\Gamma_l = QC(M_A^{may}, \varphi)$ and $\Gamma_u = QC(M_A^{must}, \varphi)$. As is the case with refinement for trigger querying, the goal of refinement is to decrease $\Gamma_u \setminus \Gamma_l$. The refinement is based on a propositional assertion θ over Φ such that $\theta \in \Gamma_u \setminus \Gamma_l$. We can choose θ arbitrarily, but typically the user provides assertions he finds interesting.

Given a formula $\theta \in \Gamma_u \setminus \Gamma_l$, there is $\xi \in \Gamma_u$ such that $\xi \rightarrow \theta$. Since $\xi \notin \Gamma_l$, it follows that $M_A^{may} \not\models \varphi[? \leftarrow \xi]$. Accordingly, refinement is similar to the one in CEGAR, which examines the counterexample for the satisfaction of $\varphi[? \leftarrow \xi]$ in M_A^{may}. Unlike CEGAR, here we refine even when the counterexample is not spurious. Indeed, predicates need to be added in order to split states along the counterexample so that the corresponding concrete computation would match a must computation in the abstraction. The process can continue until $\Gamma_l = QC(M_C, \varphi) = \Gamma_u$, but is typically terminated earlier, when the gap between Γ_l and Γ_u is of less interest.

5.2 Constrained Trigger Querying

In this variant, the input to trigger querying contains also a regular expression r over 2^P, and the set of solutions is restricted to ones in $L(r)$. Let $C_c = L_c \cap L(r)$ be the solution to the trigger querying with respect to the concrete structure. Given a set Φ of predicates, our goal is to return two sets of abstract computations that approximate C_c from below and above. Let $abs(r) = \{abs(w) : w \in L(r)\}$ and $\overline{abs}(r) = \{abs(w) : w \notin L(r)\}$. The lower and upper bounds can now be obtained by restricting L_l and L_u according to r. Formally, let $C_l = L_l \setminus \overline{abs}(r)$ and $C_u = L_u \cap abs(r)$.

Theorem 4. $C_l \subseteq_{M_C} C_c \subseteq C_u$.

We start the refinement by refining L_l and L_u. We use the algorithms described in the previous sections. In particular, we suggest to use `refineLanguage` with the input language $L(r)$. Note that it is possible for $\tau \in L_l$ to have $w, w' \in conc(\tau)$ with $w \in L(r)$ but $w' \notin L(r)$. Such computations τ are in $C_u \setminus C_l$. Let $\tau = a_1, \dots, a_n \in L_l \cap (C_u \setminus C_l)$. We continue the refinement according to the regular expression r. Since τ is a must computation, there must be at least two concrete computations $w, w' \in conc(\tau)$ such that $w \in L(r)$ and $w' \notin L(r)$. We find the first index $1 \leq i$ such that $w_i \neq w'_i$. We add a predicate that splits the abstract state a_i so that w_i and w'_i are mapped to different abstract states. We continue until $split(\tau, \Phi') \in C_l$.

5.3 Necessary Conditions

Trigger querying study sufficient conditions for φ to be triggered: if $M \models w \mapsto \varphi$, then after executing w, the suffix must satisfy φ. A dual problem is the one of finding necessary conditions for φ to hold. For a Kripke structure M and an LTL formula φ, the necessary condition for φ in M, denoted $NC(M, \varphi)$, is a set of finite computations such that for every $\pi \in L(M)$, if $\pi^n \models \varphi$ then $\pi[1..n] \in NC(M, \varphi)$. We require $NC(M, \varphi)$ to be minimal. Thus, if $w \in NC(M, \varphi)$ then there is a computation $\pi \in L(M)$ such that $\pi^{|w|} \models \varphi$ and $\pi[1, \dots, |w|] = w$.

It is shown in [21] that the problem of finding $NC(M, \varphi)$ can be solved in nondeterministic logarithmic space. Still, as in LTL model checking, abstraction would be of great help in coping with large state spaces, and as with trigger querying, we are looking for languages that approximate $NC(M, \varphi)$ from above and below. As we show below, such languages can be obtained by reasoning about the may and must abstraction of M.

Theorem 5. $NC(M_A^{must}, \varphi) \subseteq_{M_C} NC(M_C, \varphi) \subseteq NC(M_A^{may}, \varphi)$.

The refinement algorithm for necessary conditions is similar to the one used in query checking: given $\tau \in N_u \setminus N_l$, the algorithm refines both the computation τ (in case it is a may but not must computation) and the last state in it (in case it must-satisfies but does not may-satisfy φ).

6 Discussion

We described an abstraction-refinement framework for trigger querying. Beyond making trigger-querying and its variants feasible in practice, we find the framework interesting from a theoretical point of view as it involves several conceptual differences from CEGAR, and thus involves new general ideas about abstraction and refinement:

1. In CEGAR, the goal is to find a solution to a binary query: does the system satisfy the specification. Here, we sought a solution to a query that is not binary: we searched for the language L_c, and we approximated L_c from both above and below. Consequently, the lack of information in the abstraction is reflected in the distance between the approximating languages, and this distance can serve the user in the refinement process. Furthermore, termination of the procedure is determined by the user, when he finds the approximation satisfying.
2. In CEGAR, one needs to over-approximate the transitions of the system in order to reason about universal properties and to under-approximate them in order to reason about existential ones. For specification formalisms with both universal and existential path quantifiers, CEGAR needs both may and must transitions [23], and it is common to use a three-valued semantics in such cases [16]. Trigger querying does not have a universal or existential polarity, and both types of approximations are needed. However, the three-value semantics is refined to a precise measure of the lack of information, by means of $|L_u \setminus L_l|$.
3. In CEGAR, we have to use the model-checking algorithm in order to generate counterexamples, some of which may be spurious. The set of spurious counterexamples in CEGAR corresponds to the set $L_u \setminus L_l$ in our setting. Unlike the case of CEGAR, here it was possible to model this set easily by means of the automaton $\mathcal{C}$, and it was therefore possible to base the refinement process on $\mathcal{C}$. In particular, it enabled both lengthwise and widthwise refinement, and the fact the set of "counterexamples" is regular enabled a symbolic refinement procedure.

These ideas are relevant and could be helpful in several variants of CEGAR: in model checking of quantitative specifications, where the query is not binary [7], in a CEGAR method for μ-calculus, where formulas need not have a universal or existential polarity [18], in attempts to refine the three-valued semantics [3], and in algorithms that gather batches of counterexamples before refining [14,15].

References

1. Armoni, R., Fix, L., Flaisher, A., Gerth, R., Ginsburg, B., Kanza, T., Landver, A., Mador-Haim, S., Singerman, E., Tiemeyer, A., Vardi, M.Y., Zbar, Y.: The forSpec temporal logic: A new temporal property-specification logic. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 196–211. Springer, Heidelberg (2002)
2. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S.K., Ustuner, A.: Thorough static analysis of device drivers. In: EuroSys (2006)
3. Ball, T., Kupferman, O., Yorsh, G.: Abstraction for falsification. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 67–81. Springer, Heidelberg (2005)
4. Beer, I., Ben-David, S., Eisner, C., Fisman, D., Gringauze, A., Rodeh, Y.: The temporal logic sugar. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 363–367. Springer, Heidelberg (2001)
5. Bruns, G., Godefroid, P.: Temporal logic query checking. In: Proc. 16th LICS, pp. 409–420. IEEE Computer Society, Los Alamitos (2001)
6. Chan, W.: Temporal-logic queries. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 450–463. Springer, Heidelberg (2000)

7. Chatterjee, K., Doyen, L., Henzinger, T.: Quantitative languages. In: Kaminski, M., Martini, S. (eds.) CSL 2008. LNCS, vol. 5213, pp. 385–400. Springer, Heidelberg (2008)
8. Chechik, M., Gheorghiu, M., Gurfinkel, A.: Finding state solutions to temporal logic queries. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 273–292. Springer, Heidelberg (2007)
9. Chechik, M., Gurfinkel, A.: TLQSolver: A temporal logic query checker. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 210–214. Springer, Heidelberg (2003)
10. Chockler, H., Gurfinkel, A., Strichman, O.: Variants of LTL query checking. In: Raz, O. (ed.) HVC 2010. LNCS, vol. 6504, pp. 76–92. Springer, Heidelberg (2010)
11. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. Journal of the ACM 50(5), 752–794 (2003)
12. Clarke, E.M., Gupta, A., Strichman, O.: Sat-based counterexample-guided abstraction refinement. IEEE Trans. on CAD of Integrated Circuits and Systems 23(7), 1113–1123 (2004)
13. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In: Proc. 4th POPL, pp. 238–252. ACM, New York (1977)
14. de Alfaro, L., Roy, P.: Solving games via three-valued abstraction refinement. Inf. Comput. 208(6), 666–676 (2010)
15. Glusman, M., Kamhi, G., Mador-Haim, S., Fraer, R., Vardi, M.Y.: Multiple-counterexample guided iterative abstraction refinement: An industrial evaluation. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 176–191. Springer, Heidelberg (2003)
16. Godefroid, P., Jagadeesan, R.: Automatic abstraction using generalized model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 137–150. Springer, Heidelberg (2002)
17. Godefroid, P., Nori, A.V., Rajamani, S.K., Tetali, S.: Compositional must program analysis: unleashing the power of alternation. In: Proc. 37th POPL, pp. 43–56 (2010)
18. Grumberg, O., Lange, M., Leucker, M., Shoham, S.: Don't know in the μ-calculus. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 233–249. Springer, Heidelberg (2005)
19. Gurfinkel, A., Chechik, M., Devereux, B.: Temporal logic query checking: A tool for model exploration. IEEE Trans. Software Eng. 29(10), 898–914 (2003)
20. Kühne, U., Große, D., Drechsler, R.: Property analysis and design understanding. In: DATE, pp. 1246–1249 (2009)
21. Kupferman, O., Lustig, Y.: What triggers a behavior? In: Proc. 7th Int. Conf. on Formal Methods in Computer-Aided Design, pp. 146–153. IEEE Computer Society, Los Alamitos (2007)
22. Kurshan, R.P.: Computer Aided Verification of Coordinating Processes. Princeton Press, Princeton (1994)
23. Larsen, K.G., Thomsen, G.B.: A modal process logic. In: Proc. 3rd LICS (1988)
24. Lo, D., Maoz, S.: Mining scenario-based triggers and effects. In: Proc. 23rd ASE, pp. 109–118 (2008)
25. Samer, M., Veith, H.: Validity of CTL queries revisited. In: Baaz, M., Makowsky, J.A. (eds.) CSL 2003. LNCS, vol. 2803, pp. 470–483. Springer, Heidelberg (2003)
26. Vijayaraghavan, S., Ramanathan, M.: A Practical Guide for SystemVerilog Assertions. Springer, Heidelberg (2005)

Bound Analysis of Imperative Programs with the Size-Change Abstraction

Florian Zuleger[1], Sumit Gulwani[2], Moritz Sinn[1], and Helmut Veith[1,⋆]

[1] TU Wien,
{zuleger,sinn,veith}@forsyte.at
[2] Microsoft Research,
sumitg@microsoft.com

Abstract. The size-change abstraction (SCA) is an important program abstraction for termination analysis, which has been successfully implemented in many tools for functional and logic programs. In this paper, we demonstrate that SCA is also a highly effective abstract domain for the *bound analysis* of *imperative programs.*

We have implemented a bound analysis tool based on SCA for imperative programs. We abstract programs in a pathwise and context dependent manner, which enables our tool to analyze real-world programs effectively. Our work shows that SCA captures many of the essential ideas of previous termination and bound analysis and goes beyond in a conceptually simpler framework.

1 Introduction

Computing symbolic bounds for the resource consumption of imperative programs is an active area of research [15,14,13,12,11,9]. Most questions about resource bounds can be reduced to counting the number of visits to a certain program location [15]. Our research is motivated by the following technical challenges:
(A) Bounds are often *complex non-linear arithmetic expressions* built from $+, *, \max$ etc. Therefore, abstract domains based on linear invariants (e.g. intervals, octagons, polyhedra) are not directly applicable for bound computation.
(B) The proof of a given bound often requires *disjunctive invariants* that can express loop exit conditions, phases, and flags which affect program behavior. Although recent research made progress on computing disjunctive invariants [15,13,23,7,4,25,10], this is still a research challenge. (Note that the domains mentioned in (A) are conjunctive.)
(C) It is *difficult to predict a bound in terms of a template* with parameters because the search space for suitable bounds is huge. Moreover the search space cannot be reduced by compositional reasoning because bounds are global program properties.

[⋆] Research supported by the FWF research network RiSE, the WWTF grant PROSEED and Microsoft Research through a PhD Scholarship.

E. Yahav (Ed.): SAS 2011, LNCS 6887, pp. 280–297, 2011.

(D) It is not clear how to *exploit the loop structure* to achieve compositionality in the analysis for bound computation. This is in contrast to automatic termination analysis where the cutpoint technique [7,4] is used standardly.

In this paper we demonstrate that the size-change abstraction (SCA) by Lee et al. [20,3] is the right abstract domain to address these challenges. SCA is a predicate abstraction domain that consists of (in)equality constraints between integer-valued variables and boolean combinations thereof in disjunctive normal form (DNF).

SCA is well-known to be an attractive abstract domain: First, SCA is rich enough to capture the progress of many real-life programs. It has been successfully employed for automatic termination proofs of recursive functions in functional and declarative languages, and is implemented in widely used systems such as ACL2, Isabelle etc. [21,17]. Second, SCA is simple enough to achieve a good trade-off between expressiveness and complexity. For example, SCA termination is decidable and ranking functions can be extracted on terminating instances in PSPACE [3]. The simplicity of SCA sets it apart from other disjunctive abstract domains used for termination/bounds such as transition predicate abstraction [24] and powerset abstract domains [15,4].

Our method starts from the observation that progress in most software depends on the *linear change* of integer-valued functions on the program state (e.g., counter variables, size of lists, height of trees, etc.), which we call *norms*. The vast majority of non-linear bounds in real-life programs stems from two sources – nested loops and loop phases – and not from inherent non-linear behavior as in numeric algorithms. For most bounds, we have therefore the potential to exploit the nesting structure of the loops, and compose global bounds from bounds on norms. Upper bounds for norms typically consist of simple facts such as size comparisons between variables and can be computed by classical conjunctive domains. SCA is the key to convert this observation into an efficient analysis:

(1) Due to its built-in disjunctiveness and the transitivity of the order relations, SCA is closed under taking transitive hulls, and transitive hulls can be efficiently computed. We will use this for summarizing inner loops.

(2) We use SCA to compose global bounds from bounds on the norms. To extract norms from the program, we only need to consider small program parts. After the (local) extraction we have to consider only the size-change-abstracted program for bound computation.

(3) SCA is the natural abstract domain in connection with two program transformations – *pathwise analysis* and *contextualization* – that make imperative programs more amenable to bound analysis. Pathwise analysis is used for reasoning about complete program paths, where inner loops are overapproximated by their transitive hulls. Contextualization adds path-sensitivity to the analysis by checking which transitions can be executed subsequently. Both transformations make use of the progress in SMT solver technology to reason about the long pieces of straight-line code given by program paths.

Summary of our Approach. To determine how often a location l of program P can be visited, we proceed in two steps akin to [15]: First, we compute a disjunctive transition system $\mathcal{T}$ for l from P. Second, we use $\mathcal{T}$ to compute a bound on the number of visits to l. For the first step we recursively compute transition systems for nested loops and summarize them disjunctively by transitive hulls computed with SCA. We enumerate all cycle-free paths from l back to l, and derive a disjunctive transition system $\mathcal{T}$ from these paths and the summaries of the inner loops using *pathwise analysis*. For the second step we exploit the potential of SCA for automatic bound computation by first abstracting $\mathcal{T}$ using norms extracted from the program and then computing bounds solely on the abstraction. We use *contextualization* to increase the precision of the bound computation. Our method thus clearly addresses the challenges **(A)** to **(D)** discussed above. In particular, we make the following new contributions:

- We are the first to exploit SCA for bound analysis by using its ability of composing global bounds from bounds on locally extracted norms and disjunctive reasoning. Our technical contributions are the first algorithm for computing bounds with SCA (Algorithm 2) and the disjunctive summarization of inner loops with SCA (Algorithm 1).
- We are the first to describe how to apply SCA on imperative programs. Our technical contributions are two program transformations: pathwise analysis (Subsection 5.2), which exploits the looping structure of imperative programs, and contextualization (Subsection 6.1). These program transformations make imperative programs amenable to bound analysis by SCA.
- We obtain a competitive bound algorithm that captures the essential ideas of earlier termination and bound analyses in a simpler framework. Since bound analysis generalizes termination analysis, many of our methods are relevant for termination. Our experimental section shows that we can handle a large percentage of standard C programs. We give a detailed comparison with related work on termination and bound analysis in the extended version of this paper available on the website of the first author.

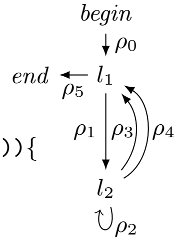
2 Examples

We use two examples to demonstrate the challenges in the automatic generation of transition systems and bound computation, and give an overview of our approach. In the examples, we denote transition relations as expressions over primed and unprimed state variables in the usual way.

Example 1: Transition System Generation. Let us consider the source code of Example 1 together with its (simplified) CFG and transition relations in Figure 1. Computing a bound for the header of the outer loop l_1 exhibits the following difficulties: The inner loop cannot be excluded in the analysis of the outer loop (e.g. by the standard technique called *slicing*) as it modifies the counter of the outer loop; this demonstrates the need for global reasoning in bound analysis. Further one needs to distinguish whether the inner loop has been skipped or executed at least one time as this determines whether $j = 0$ or $j > 0$. This exemplifies why we need disjunctive invariants for inner loops.

Example 1.

```
     void main (int n){
       int i = 0; int j;
l1:    while(i < n) {
         i++; j := 0;
l2:      while((i < n) && ndet()){
           i++; j++; }
         if (j > 0)
           i--; } }
```



$$\rho_0 \equiv i = 0$$
$$\rho_1 \equiv i < n \wedge i' = i + 1 \wedge j' = 0$$
$$\rho_2 \equiv i < n \wedge i' = i{+}1 \wedge j' {=} j + 1$$
$$\rho_3 \equiv j > 0 \wedge i' = i - 1$$
$$\rho_4 \equiv j \leq 0$$
$$\rho_5 \equiv i \geq n$$

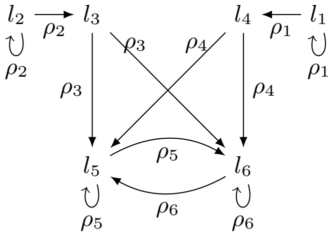
Fig. 1. Example 1 with its (simplified) CFG and transition relations

Moreover, the counter i may decrease, but this can only happen when i has been increased by at least 2 before. This presents a difficulty to an automatic analysis since the used abstract domains need to be precise enough to capture such reasoning. In particular, a naive application of the size-change abstraction is too imprecise, since it contains only inequalities.

Our Algorithm 1 computes a transition system for the outer loop with header l_1 as follows: The algorithm is based on the idea of enumerating all paths from l_1 back to l_1 in order to derive a precise disjunctive transition system. However, this enumeration is not possible as there are infinitely many such paths because of the inner loop at l_2. Therefore Algorithm 1 recursively computes a transition system $\{i < n \wedge i' = i + 1 \wedge j' = j + 1 \wedge n' = n\}$ for the inner loop at l_2, and then summarizes the inner loop disjunctively by size-change abstracting its transition system to $\{n - i > 0 \wedge n' - i' < n - i \wedge j < j'\}$ (our analysis extracts the norms $n{-}i, j$ from the program using heuristics, cf. Section 7) and computing the reflexive transitive hull $\{n' - i' = n - i \wedge j' = j, n - i > 0 \wedge n' - i' < n - i \wedge j < j'\}$ in the abstract. (Note that we use sets of formulae to denote disjunctions of formulae.) Then Algorithm 1 enumerates all cycle-free paths from l_1 back to l_1. There are two such paths: $\pi_1 = l_1 \xrightarrow{\rho_1} l_2 \xrightarrow{\rho_3} l_1$ and $\pi_2 = l_1 \xrightarrow{\rho_1} l_2 \xrightarrow{\rho_4} l_1$. Algorithm 1 inserts the reflexive transitive hull $\mathcal{T}$ of the inner loop on the paths π_1, π_2 at the header of the inner loop l_2 and contracts the transition relations. This results in the two transition relations $\{\mathit{false}, n - i - 1 > 0 \wedge n' - i' < n - i \wedge j' > 0\}$ for π_1 (one for each disjunct of the summary of the inner loop), and $\{n - i > 0 \wedge n' - i' = n - i - 1 \wedge j' = 0, \mathit{false}\}$ for π_2. Note that for each path, *false* indicates that one transition relation was detected to be unsatisfiable, e.g. $n - i - 1 > 0 \wedge n' - i' < n - i - 1 \wedge j' > 0 \wedge j' \leq 0$ in π_2. Algorithm 1 returns the satisfiable two transitions as a transition system $\mathcal{T}$ for the outer loop.

Our Algorithm 2 size-change abstracts $\mathcal{T}$ (resulting in $\{n - i > 0 \wedge n' - i' < n - i \wedge j' > 0, n - i > 0 \wedge n' - i' < n - i \wedge j' >= 0\}$) and computes the bound $\max(n, 0)$ from the abstraction. The difficult part in analyzing Example 1 is the transition system generation, while computing a bound from $\mathcal{T}$ is easy.

Example 2: Bound Computation. Bound analysis is complicated when a loop contains a finite state machine that controls its dynamics. Example 2, found during our experiments on the `cBench` benchmark [1], presents such a loop.



The size-change abstractions of the transition relations:

ρ_1: $l > 0 \wedge s' > s \wedge s' \leq 255 \wedge l' = l$
ρ_2: $l > 0 \wedge s' < s \wedge s' \geq 0 \wedge l' = l$
ρ_3 and ρ_5: $l > 0 \wedge l' < l \wedge s' > s \wedge s' \leq 255$
ρ_4 and ρ_6: $l > 0 \wedge l' < l \wedge s' < s \wedge s' \geq 0$

Fig. 2. The CFG obtained from contextualizing the transition system of Example 2 (left) and the size-change abstractions of the transition relations (right)

Example 2. `// cBench/consumer_lame/src/quantize-pvt.c`

```
int bin_search_StepSize2 (int r, int s) {
  static int c = 4; int n; int f = 0; int d = 0;
  do {
    n = nondet();
    if (c == 1 ) break;
    if (f) c /= 2;
    if (n > r) {
      if (d == 1 && !f) {f = 1; c /= 2; }
      d = 2; s += c;
      if (s > 255) break; }
    else if (n < r) {
      if (d == 2 && !f) {f = 1; c /= 2; }
      d = 1; s -= c;
      if (s < 0) break; }
    else break; }
  while (1); }
```

The loop has three different phases: in its first iteration it assigns 1 or 2 to d, then either increases or decreases s until it sets f to true; then it divides c by 2 until the loop is exited. Note that disjunctive reasoning is crucial to distinguish the phases!

Our method first uses a standard invariant analysis (such as the octagon analysis) to compute the invariant $c \geq 1$, which is valid throughout the execution of the loop. Then Algorithm 1 obtains a transition system from the loop by collecting all paths from loop header back to the loop header. Omitting transitions that belong to infeasible paths we obtain six transitions:

$$\rho_1 \equiv c \geq 1 \wedge \neg f \wedge d \neq 1 \wedge d' = 2 \wedge s' = s + c \wedge s' \leq 255 \wedge c' = c \wedge f' = f$$
$$\rho_2 \equiv c \geq 1 \wedge \neg f \wedge d \neq 2 \wedge d' = 1 \wedge s' = s - c \wedge s' \geq 0 \wedge c' = c \wedge f' = f$$
$$\rho_3 \equiv c \geq 1 \wedge \neg f \wedge d = 1 \wedge f' \wedge c' = c/2 \wedge d' = 2 \wedge s' = s + c' \wedge s' \leq 255$$
$$\rho_4 \equiv c \geq 1 \wedge \neg f \wedge d = 2 \wedge f' \wedge c' = c/2 \wedge d' = 1 \wedge s' = s - c' \wedge s' \geq 0$$
$$\rho_5 \equiv c \geq 1 \wedge f \wedge c' = c/2 \wedge d' = 2 \wedge s' = s + c' \wedge s' \leq 255 \wedge f' = f$$
$$\rho_6 \equiv c \geq 1 \wedge f \wedge c' = c/2 \wedge d' = 1 \wedge s' = s - c' \wedge s' \geq 0 \wedge f' = f$$

Our bound analysis reasons about this transition system automatically by applying the program transformation called *contextualization*, which determines in which context transitions can be executed, and size-change abstracting the

transitions. By our heuristics (cf. Section 7) we consider s and the logarithm of c (which we abbreviate by l) as program norms.
Figure 2 shows the CFG obtained from contextualizing the transition system of Example 2 on the left. The CFG vertices carry the information which transition is executed next. The CFG edges are labeled by the transitions of the transition system, where presence of edges indicates that, e.g., l_4 can be directly executed after l_1, and absence of an arc from l_4 to l_1 means that this transition is infeasible. The CFG shows that the transitions cannot interleave in arbitrary order; particularly useful are the strongly-connected components (SCCs) of the CFG. Our bound Algorithm 2 exploits the SCC decomposition. It computes bounds for every SCC separately using the size-change abstracted transitions (cf. Figure 2 on the right) and composes them to the overall bound $\max(255, s) + 3$, which is precise.

We point out how the above described approach *enables* automatic bound analysis by SCA. Note that the variables d and f do not appear in the abstracted transitions. It is sufficient for our analysis to work with the CFG obtained from contextualization because the loop behavior of Example 2, which is controlled by d and f, has been encoded into the CFG. This has the advantage that less variables have to be considered in the actual bound analysis. Further note that the CFG decomposition gives us compositionality in bound analysis. Our analysis is able to combine the bounds of the SCCs to an (fairly complicated) overall bound using the operators max and + by following the structure of the CFG.

3 Program Model and Size-Change Abstraction

Sets and Relations. Let A be a set. The concatenation of two relations $B_1, B_2 \in 2^{A\times A}$ is the relation $B_1 \circ B_2 = \{(e_1, e_3) \mid \exists e_2.(e_1, e_2) \in B_1 \wedge (e_2, e_3) \in B_2\}$. $Id = \{(e, e) \mid e \in A\}$ is the *identity relation* over A. Let $B \in 2^{A\times A}$ be a relation. We inductively define the *k-fold exponentiation* of B by $B^k = B^{k-1} \circ B$ and $B^0 = Id$. $B^+ = \bigcup_{k\geq 1} B^k$ resp. $B^* = \bigcup_{k\geq 0} B^k$ is the *transitive*- resp. *reflexive transitive hull* of B. We lift the concatenation operator $\circ$ to sets of relations by defining $\mathcal{C}_1 \circ \mathcal{C}_2 = \{B_1 \circ B_2 \mid B_1 \in \mathcal{C}_1, B_2 \in \mathcal{C}_2\}$ for sets of relations $\mathcal{C}_1, \mathcal{C}_2 \subseteq 2^{A\times A}$. We set $\mathcal{C}^0 = \{Id\}$; $\mathcal{C}^k, \mathcal{C}^+$ etc. are defined analogously.

Program Model. We introduce a simple program model for sequential imperative programs without procedures. Our definition models explicitly the essential features of imperative programs, namely branching and looping. In Section 5 we will explain how to exploit the graph structure of programs in our analysis algorithm. We leave the extension to concurrent and recursive programs for future work.

Definition 1 (Transition Relations / Invariants). *Let Σ be a set of* states. *The set of* transition relations $\Gamma = 2^{\Sigma\times\Sigma}$ *is the set of relations over Σ. A* transition set $\mathcal{T} \subseteq \Gamma$ *is a finite set of transition relations. Let $\rho \in \Gamma$ be a transition relation. $\mathcal{T}$ is a* transition system *for ρ, if $\rho \subseteq \bigcup \mathcal{T}$. $\mathcal{T}$ is a* transition invariant *for ρ, if $\rho^* \subseteq \bigcup \mathcal{T}$.*

Definition 2 (Program, Path, Trace, Termination). *A* program *is a tuple $P = (L, E)$, where L is a finite set of* locations*, and $E \subseteq L \times \Gamma \times L$ is a finite set of* transitions*. We write $l_1 \xrightarrow{\rho} l_2$ to denote a transition (l_1, ρ, l_2).*

A path *of P is a sequence $l_0 \xrightarrow{\rho_0} l_1 \xrightarrow{\rho_1} \cdots$ with $l_i \xrightarrow{\rho_i} l_{i+1} \in E$ for all i. Let $\pi = l_0 \xrightarrow{\rho_0} l_1 \xrightarrow{\rho_1} l_2 \cdots l_k \xrightarrow{\rho_k} l_{k+1}$ be a finite path. π is* cycle-free*, if π does not visit a location twice except for the end location, i.e., $l_i \neq l_j$ for all $0 \leq i < j \leq k$. The* contraction *of π is the transition relation $\mathtt{rel}(\pi) = \rho_0 \circ \rho_1 \circ \cdots \circ \rho_k$ obtained from concatenating all transition relations along π. Given a location l, $\mathtt{paths}(P, l)$ is the set of all finite paths with start and end location l. A path $\pi \in \mathtt{paths}(P, l)$ is* simple*, if all locations, except for the start and end location, are different from l.*

A trace *of P is a sequence $(l_0, s_0) \xrightarrow{\rho_0} (l_1, s_1) \xrightarrow{\rho_1} \cdots$ such that $l_0 \xrightarrow{\rho_0} l_1 \xrightarrow{\rho_1} \cdots$ is a path of P, $s_i \in \Sigma$ and $(s_i, s_{i+1}) \in \rho_i$ for all i. P is* terminating*, if there is no infinite trace of P.*

Note that a cycle-free path $\pi \in \mathtt{paths}(P, l)$ is always simple. Further note that our definition of programs allows to model branching and looping precisely and naturally: imperative programs can usually be represented as CFGs whose edges are labeled with assign and assume statements.

Definition 3 (Transition Relation of a Location). *Let $P = (L, E)$ be a program and $l \in L$ a location. The* transition relation of l *is the set $P|_l = \bigcup_{simple\ \pi \in \mathtt{paths}(P,l)} \mathtt{rel}(\pi)$.*

3.1 Order Constraints

Let X be a set of variables. Given a variable x we denote by x' its *primed* version. We denote by X' the set $\{x' \mid x \in X\}$ of the primed variables of X. We denote by $\triangleright$ any element from $\{>, \geq\}$.

Definition 4 (Order Constraint). *An* order constraint *over X is an inequality $x \triangleright y$ with $x, y \in X$.*

Definition 5 (Valuation). *The set of all* valuations *of X is the set $Val_X = X \to \mathbb{Z}$ of all functions from X to the integers. Given a valuation $\sigma \in Val_X$ we define its* primed valuation *as the function $\sigma' \in Val_{X'}$ with $\sigma'(x') = \sigma(x)$ for all $x \in X$. Given two valuations $\sigma_1 \in Val_{X_1}, \sigma_2 \in Val_{X_2}$ with $X_1 \cap X_2 = \emptyset$ we define their* union *$\sigma_1 \cup \sigma_2 \in Val_{X_1 \cup X_2}$ by $(\sigma_1 \cup \sigma_2)(x) = \begin{cases} \sigma_1(x) & for\ x \in X_1, \\ \sigma_2(x) & for\ x \in X_2. \end{cases}$*

Definition 6 (Semantics). *We define a* semantic relation *$\models$ as follows: Let $\sigma \in Val_X$ be a valuation. Given an order constraint $x_1 \triangleright x_2$ over X, $\sigma \models x_1 \triangleright x_2$ holds, if $\sigma(x_1) \triangleright \sigma(x_2)$ holds in the structure of the integers $(\mathbb{Z}, \geq)$. Given a set O of order constraints over X, $\sigma \models O$ holds, if $\sigma \models o$ holds for all $o \in O$.*

3.2 Size-Change Abstraction (SCA)

We are using integer-valued functions on the program states to measure progress of a program. Such functions are called norms in the literature. Norms provide us sizes of states that we can compare. We will use norms for abstracting programs.

Definition 7 (Norm). *A* norm $n \in \Sigma \to \mathbb{Z}$ *is a function that maps the states to the integers.*

We fix a finite set of norms N for the rest of this subsection, and describe in Section 7 how to extract norms from programs automatically. Given a state $s \in \Sigma$ we define a valuation $\sigma_s \in Val_N$ by setting $\sigma_s(n) = n(s)$.

We will now introduce SCA. Our terminology diverts from the seminal papers on SCA [20,3] because we focus on a logical rather than a graph-theoretic representation. The set of norms N corresponds to the SCA "variables" in [20,3].

Definition 8 (Monotonicity Constraint, Size-change Relation / Set, Concretization). *The set of* monotonicity constraints *MCs is the set of all order constraints over* $N \cup N'$. *The set of* size-change relations *(SCRs)* $SCRs = 2^{MCs}$ *is the powerset of MCs. An* SCR set $\mathcal{S} \subseteq SCRs$ *is a set of SCRs. We use the* concretization function $\gamma : SCRs \to \Gamma$ *to map an SCR* $T \in SCRs$ *to a transition relation* $\gamma(T)$ *by defining* $\gamma(T) = \{(s_1, s_2) \in \Sigma \times \Sigma \mid \sigma_{s_1} \cup \sigma'_{s_2} \models T\}$ *as the set of all pairs of states such that the evaluation of the norms on these states satisfy all the constraints of* T. *We lift the concretization function to SCR sets by setting* $\gamma(\mathcal{S}) = \{\gamma(T) \mid T \in \mathcal{S}\}$ *for an SCR set* $\mathcal{S}$.

Note that the abstract domain of SCRs has only finitely many elements, namely $3^{(2|N|)^2}$. Further note that an SCR set corresponds to a formula in DNF.

Definition 9 (Abstraction Function). *The* abstraction function $\alpha : \Gamma \to SCRs$ *takes a transition relation* $\rho \in \Gamma$ *and returns the greatest SCR containing it, namely* $\alpha(\rho) = \{c \in MCs \mid \rho \subseteq \gamma(c)\}$. *We lift the abstraction function to transition sets by setting* $\alpha(\mathcal{T}) = \{\alpha(\rho) \mid \rho \in \mathcal{T}\}$ *for a transition set* $\mathcal{T}$.

Implementation of the abstraction. α can be implemented by an SMT solver under the assumption that the norms are provided as expressions and that the transition relation is given as a formula such that the order constraints between these expressions and the formula fall into a logic that the SMT solver can decide.

Using abstraction and concretization we can define concatenation of SCRs:

Definition 10 (Concatenation of SCRs). *Given two SCRs* $T_1, T_2 \in SCRs$, *we define* $T_1 \circ T_2$ *to be the SCR* $\alpha(\gamma(T_1) \circ \gamma(T_2))$. *We lift the concatenation operator* $\circ$ *to SCR sets by defining* $\mathcal{S}_1 \circ \mathcal{S}_2 = \{T_1 \circ T_2 \mid T_1 \in \mathcal{S}_1, T_2 \in \mathcal{S}_2\}$ *for SCR sets* $\mathcal{S}_1, \mathcal{S}_2 \in 2^{SCRs}$. $\mathcal{S}^0 = \{Id\}, \mathcal{S}^k, \mathcal{S}^+, \mathcal{S}^*$ *etc. are defined in the natural way.*

Concatenation of SCRs is conservative by definition, i.e., $\gamma(T_1 \circ T_2) \supseteq \gamma(T_1) \circ \gamma(T_2)$ and associative because of the transitivity of order relations. Concatenation of SCRs can be effectively computed by a modified all-pairs-shortest-path algorithm (taking order relations as weights). Because the number of SCRs is finite, the transitive hull is computable.

The following theorem can be directly shown from the definitions. We will use it to summarize the transitive hull of loops disjunctively, cf. Section 5.

Theorem 1 (Soundness). *Let ρ be a transition relation and $\mathcal{T}$ a transition system for ρ. Then $\gamma(\alpha(\mathcal{T})^*)$ is a transition invariant for ρ.*

4 Main Steps of our Analysis

Let $P = (L, E)$ be a program and $l \in L$ be a location for which we want to compute a bound. Our analysis consists of four main steps:

1. Extract a set of norms N using heuristics (Section 7)
2. Compute global invariants by standard abstract domains
3. Compute $\mathcal{T} = \texttt{TransSys}(P, l)$ (Section 5)
4. Compute $b = \texttt{Bound}(\texttt{Contextualize}(\mathcal{T}))$ (Section 6)

In Step 1 we extract a set of norms N using the heuristics described in Section 7. The abstraction function α that we use in Steps 3 and 4 is parameterized by the set of norms N. In Step 2 we compute global invariants by standard abstract domains such as interval, octagon or polyhedra. As this step is standard, we do not discuss it in this paper. In Step 3 we compute a transition system $\mathcal{T} = \texttt{TransSys}(P, l)$ for $P|_l$ by Algorithm 1. In Step 4 we compute a bound $b = \texttt{Bound}(\texttt{Contextualize}(\mathcal{T}))$ for the number of visits to l, where we first use the program transformation contextualization of Definition 11 to transform $\mathcal{T}$ into a program from which we then compute a bound b by Algorithm 2.

```
Procedure: TransSys$(P, l)$
Input: a program $P = (L, E)$, a location $l \in L$
Output: a transition system for $P|_l$
Global: array summary for storing transition invariants
foreach $(loop, header) \in NestedLoops(P, l)$ do
    $\mathcal{T}$ := TransSys$(loop, header)$;
    $hull := \gamma(\alpha(\mathcal{T})^*)$;
    summary$[header] := hull$;
foreach cycle-free path $\pi = l \xrightarrow{\rho_0} l_1 \xrightarrow{\rho_1} l_2 \cdots l_k \xrightarrow{\rho_k} l \in$ paths$(P, l)$ do
    $\mathcal{T}_\pi := \{\rho_0\} \circ$ ITE(IsHeader$(l_1)$, summary$[l_1]$, $\{Id\}) \circ \{\rho_1\} \circ$
                      ITE(IsHeader$(l_2)$, summary$[l_2]$, $\{Id\}) \circ \{\rho_2\} \circ \cdots \circ$
                      ITE(IsHeader$(l_k)$, summary$[l_k]$, $\{Id\}) \circ \{\rho_k\}$;
return $\bigcup_{\text{cycle-free path } \pi \in \texttt{paths}(P,l)} \mathcal{T}_\pi$;
```

Algorithm 1. $\texttt{TransSys}(P, l)$ computes a transition system for $P|_l$

5 Computing Transition Systems

In this section we describe our algorithm for computing transition systems. We first present the actual algorithm, and then discuss specific characteristics. The function `TransSys` in Algorithm 1 takes as input a program $P = (L, E)$ and a location $l \in L$ and computes a transition system for $P|_l$, cf. Theorem 2 below. The key ideas of Algorithm 1 are (1) to summarize inner loops disjunctively by transition invariants computed with SCA, and (2) to enumerate all cycle-free paths for pathwise analysis. Note that for loop summarization the algorithm is recursively invoked. We give an example for the application of Algorithm 1 to Example 1 in the extended version.

Loop Summarization. In the first `foreach`-loop, Algorithm 1 iterates over all nested loops of P w.r.t. l. A loop *loop* of P is a nested loop w.r.t. l, if it is strongly connected to l but does not contain l, and if there is no loop with the same properties that strictly contains *loop*. Let *loop* be a nested loop of P w.r.t. l and let *header* be its header. (We assume that the program is reducible, see discussion below.) `TransSys` calls itself recursively to compute a transition system $\mathcal{T}$ for $loop|_{header}$.

In statement $hull := \gamma(\alpha(\mathcal{T})^*)$, $\alpha(\mathcal{T})$ size-change abstracts $\mathcal{T}$ to an SCR set, $\alpha(\mathcal{T})^*$ computes the transitive hull of this SCR set, and $\gamma(\alpha(\mathcal{T})^*)$ concretizes the abstract transitive hull to a transition set, which is then assigned to *hull*. Algorithm 1 stores *hull* in the array $\mathsf{summary}$, which is a transition invariant for $loop|_{header}$ by the soundness of SCA as stated in Theorem 1.

After the first `foreach`-loop, Algorithm 1 has summarized all inner loops, not only the nested loops, because the recursive calls reaches all nesting levels. For each inner loop *loop* with header *header* a transition invariant for $loop|_{header}$ has been stored at $\mathsf{summary}[header]$. Summaries of inner loops are visible to all outer loops, because the array $\mathsf{summary}$ is a global variable.

Pathwise Analysis. In the second `foreach`-loop, Algorithm 1 iterates over all cycle-free paths of P with start and end location l. Let $\pi = l \xrightarrow{\rho_0} l_1 \xrightarrow{\rho_1} \cdots l_k \xrightarrow{\rho_k} l$ be such a cycle-free path. The expression $\mathtt{ITE}(\mathtt{IsHeader}(l_i), \mathsf{summary}[l_i], \{Id\})$ evaluates to $\mathsf{summary}[l_i]$ for each location l_i, if l_i is the header of an inner loop $loop_i$, and evaluates to the transition set $\{Id\}$, which contains only the identity relation over the program states, else. Algorithm 1 computes the set $\mathcal{T}_\pi = \{\rho_0\} \circ$
$\mathtt{ITE}(\mathtt{IsHeader}(l_1), \mathsf{summary}[l_1],$
$\{Id\}) \circ \{\rho_1\} \circ \mathtt{ITE}(\mathtt{IsHeader}(l_2), \mathsf{summary}[l_2], \{Id\})$
$\circ \{\rho_2\} \circ \cdots \circ \mathtt{ITE}(\mathtt{IsHeader}(l_k), \mathsf{summary}[l_k], \{Id\}) \circ \{\rho_k\}$, which is an overapproximation of the contraction of π, where the summaries of the inner loops $loop_i$ are inserted at their headers l_i. The transition set $\mathcal{T}_\pi$ overapproximates all paths starting and ending in l that iterate arbitrarily often through inner loops along π, because for every loop $loop_i$ the transition set $\mathsf{summary}[l_i]$ overapproximates all paths starting and ending in l_i that iterate arbitrarily often through $loop_i$ (as $\mathsf{summary}[l_i]$ is a transition invariant for $loop_i|_{l_i}$). Algorithm 1 returns the union $\bigcup_{\text{cycle-free path } \pi \in \mathtt{paths}(P,l)} \mathcal{T}_\pi$ of all those transition sets $\mathcal{T}_\pi$.

Theorem 2. *Algorithm 1 computes a transition system* $\mathtt{TransSys}(P, l)$ *for* $P|_l$.

Proof (Sketch). Let $\pi' \in \mathtt{paths}(P, l)$ be a simple path. We obtain a cycle-free path $\pi \in \mathtt{paths}(P, l)$ from π' by deleting all iterations through inner loops of (P, l) from π'. The transition set $\mathcal{T}_\pi$ overapproximates all paths starting and ending in l that iterate arbitrarily often through inner loops of (P, l) along π. As π' iterates through inner loops of (P, l) along π we have $\mathtt{rel}(\pi) \subseteq \bigcup \mathcal{T}_\pi$.

Implementation. We use conjunctions of formulae to represent individual transitions. This allows us to implement the concatenation of transition relations by conjoining their formulae and introducing existential quantifiers for the intermediate variables. We detect empty transition relations by asking an SMT solver whether their corresponding formulae are satisfiable. We use these emptiness checks at several points during the analysis to reduce the number of transition relations.

Algorithm 1 may exponentially blow up in size because of the enumeration of all cycle-free paths and the computation of transitive hulls of inner loops. We observed in our experiments that by first extracting norms from the program under scrutiny and then slicing the program w.r.t. these norms before continuing with the analysis normally results into programs small enough for making our analysis feasible.

Irreducible programs. Algorithm 1 refers to loop headers, and thus implicitly assumes that loops are reducible. (Recall that in a reducible program each SCC has a unique entry point called the header.) We have formulated Algorithm 1 in this way to make clear how it exploits the looping structure of imperative programs. However, Algorithm 1 can be easily extended to irreducible loops by a case distinction on the (potentially multiple) entry points of SCCs.

5.1 Disjunctiveness in Algorithm 1

Disjunctiveness is crucial for bound analysis. We have given two examples for this fact in Section 2 and refer the reader for further examples to [15,4,23]. We emphasize that our analysis can handle *all examples* of these publications. Our analysis is disjunctive in two ways:

(1) *We summarize inner loops disjunctively.* Given a transition system $\mathcal{T}$ for some inner loop *loop*, we want to summarize *loop* by a transition invariant. The most precise transition invariant $\mathcal{T}^* = \{Id\} \cup \mathcal{T} \cup \mathcal{T}^2 \cup \mathcal{T}^3 \cup \cdots$ introduces infinitely many disjunctions and is not computable in general. In contrast to this the abstract transitive hull $\alpha(\mathcal{T})^* = \alpha(\{Id\}) \cup \alpha(\mathcal{T}) \cup \alpha(\mathcal{T})^2 \cup \alpha(\mathcal{T})^3 \cup \cdots$ has only finitely many disjunctions and is effectively computable. This allows us to overapproximate the infinite disjunction $\mathcal{T}^*$ by the finite disjunction $\gamma(\alpha(\mathcal{T})^*)$.

We underline that the need for disjunctive summaries of inner loops in the bound analysis is a major motivation for SCA, as it allows us to compute disjunctive transitive hulls naturally, cf. definition and discussion in Section 3.2.

(2) *We summarize local transition relations disjunctively.* Given a program $P = (L, E)$ and location $l \in L$, we want to compute a transition system for $P|_l$.

For a cycle-free path $\pi \in \texttt{paths}(P, l)$ the transition set $\mathcal{T}_\pi$ computed in Algorithm 1 overapproximates all simple paths in $\texttt{paths}(P, l)$ that iterate through inner loops along π. As all $\mathcal{T}_\pi$ are sets, the set union $\bigcup_{\text{cycle-free path } \pi \in \texttt{paths}(P,l)} \mathcal{T}_\pi$ is a disjunctive summarization of all $\mathcal{T}_\pi$ that keeps the information from different paths separated. This is important for our analysis which relies on the observation that monotonic changes of norms can be observed along single paths from loop header back to the header.

5.2 Pathwise Analysis in Algorithm 1

It is well-known that analyzing large program parts jointly improves the precision of static analyses, e.g. [6]. Owing to the progress in SMT solvers this idea has recently seen renewed interested by static analyses such as abstract interpretation [22] and software model checking [5], which use SMT solvers for abstracting large blocks of straight-line code jointly to increase the precision of the analysis.

We call the analyses of [22,5] and classical SCA [20,3] *blockwise*, because they do joint abstraction only for loop-free program parts. In contrast, our *pathwise analysis* abstracts complete paths at once: Algorithm 1 enumerates all cycle-free paths from loop header to loop header and inserts summaries for inner loops on these paths. These paths are then abstracted jointly in a subsequent loop summarization or bound computation. In this way our pathwise analysis is strictly more precise than blockwise analysis. We illustrate this issue on Example 1 for SCA the extended version of this paper.

Parsers are a natural class of programs which illustrate the need for pathwise analysis. In our experiments we observed that many parsers increase an index while scanning the input stream and use lookahead to detect which token comes next. As in Example 1, lookaheads may temporarily decrease the index. Pathwise abstraction is crucial to reason about the program progress with SCA.

6 Bound Computation

Our bound computation consists of two steps. Step 1 is the program transformation contextualization which transforms a transition system into a program. Step 2 is the bound algorithm which computes bounds from programs.

6.1 Contextualization

Contextualization is a program transformation by Manolios and Vroon [21], who report on an impressive precision of their SCA-based termination analysis of functional programs. Note that we do not use their terminology (e.g. “calling context graphs”) in this paper. Our contribution lies in adopting contextualization to imperative programs and in recognizing its relevance for bound analysis.

Definition 11 (Contextualization). *Let $\mathcal{T}$ be a transition set. The* contextualization *of $\mathcal{T}$ is the program $P = (\mathcal{T}, E)$, where $E = \{\rho \xrightarrow{\rho} \rho' \mid \rho, \rho' \in \mathcal{T}$ and $\rho \circ \rho' \neq \emptyset\}$.*

Example 3.

```
void main (int x, int b){
while (0 < x < 255){
  if (b) x = x + 1;
  else x = x - 1; } }
```

$$\rho_1 \equiv 0 < x < 255 \wedge b \wedge x' = x + 1 \wedge b'$$
$$\rho_2 \equiv 0 < x < 255 \wedge \neg b \wedge x' = x - 1 \wedge \neg b'$$

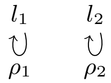


Fig. 3. Example 3 with its transition relations and CFG obtained from contextualization

The contextualization of a transition system is a program in which every location determines which transition is executed next; the program has an edge between two locations only if the transitions of the locations can be executed one after another.

Contextualization restricts the order in which the transitions of the transition system can be executed. Thus, contextualization encodes information that could otherwise be deduced from the pre- and postconditions of transitions directly into the CFG. Since pathwise analysis contracts whole loop paths into single transitions, contextualization is particularly important after pathwise analysis: our subsequent bound algorithm does not need to compute the pre- and postcondition of the contracted loop paths but only needs to exploit the structure of the CFGs for determining in which order the loop paths can be executed.

We illustrate contextualization on Example 3. The program has two paths, and gives rise to the transition system $\mathcal{T} = \{\rho_1, \rho_2\}$. Keeping track of the boolean variable b is important for bound analysis: Without reference to b not even the termination of `main` can be proven. In Figure 3 (right) we show the contextualization of $\mathcal{T}$. Note that contextualization has encoded information about the variable b into the CFG in such a way that we do not need to keep track of the variable b anymore. Thus, contextualization releases us from taking the precondition b resp. $\neg b$ and the postcondition b' resp. $\neg b'$ into account for bound analysis.

At the beginning we gave an application of contextualization on the sophisticated loop in Example 2, where contextualization uncovers the control structure of the finite state machine encoded into the loop. An application of contextualization to the flagship example of a recent publication [13] can be found in the extended version of this paper.

Note that in our definition of contextualization we only consider the consistency of two consecutive transitions. It would also have been possible to consider three or more consecutive transitions. This would result in increased precision. However, we found two transitions to be sufficient in practice.

Implementation. We implement contextualization by encoding the concatenation $\rho_1 \circ \rho_2$ of two transitions ρ_1, ρ_2 into a logical formula and asking an SMT solver whether this formula is satisfiable. Note that such a check is very simple to implement in comparison to the explicit computation of pre- and postconditions.

```
Procedure: Bound($P$)
Input: a program $P = (L, E)$
Output: a bound $b$ on the length of the traces of $P$
$SCCs$ := computeSCCs($P$); $b := 0$;
while $SCCs \neq \emptyset$ do
    $SCCsOnLevel := \emptyset$;
    forall $SCC \in SCCs$ s.t. no $SCC' \in SCCs$ can reach $SCC$ do
        $r :=$ BndSCC($SCC$);
        Let $r \leq b_{SCC}$ be a global invariant;
        $SCCsOnLevel := SCCsOnLevel \cup \{SCC\}$;
    $b := b + \max_{SCC \in SCCsOnLevel} b_{SCC}$;
    $SCCs := SCCs \setminus SCCsOnLevel$;
return $b$;
```

Algorithm 2. Bound composes the bounds of the SCCs to an overall bound

6.2 Bound Algorithm

Our bound algorithm reduces the task of bound computation to the computation of local bounds and the composition of these local bounds to an overall bound. To this end, we exploit the structure of the CFGs obtained from contextualization: We partition the CFG of programs into its strongly connected components (SCCs) (SCCs are maximal strongly connected subgraphs). For each SCC, we compute a bound by Algorithm 3, and then compose these bounds to an overall bound by Algorithm 2.

Algorithm 2 arranges the SCCs of the CFG into levels: The first level consists of the SCCs that do not have incoming edges, the second level consists of the SCCs that can be reached from the first level, etc. For each level, Algorithm 2 calls Algorithm 3 to compute bounds for the SCCs of this level. Let SCC be an SCC of some level and let $r :=$ BndSCC(SCC) be the bound returned by Algorithm 3 on SCC. r is a (local) bound of SCC that may contain variables of P that are changed during the execution of P. Algorithm 2 uses global invariants (e.g. interval, octagon or polyhedra) in order to obtain a bound b_{SCC} on r in terms of the initial values of P. The SCCs of one level are collected in the set *SCCsOnLevel*. For each level, Algorithm 2 composes the bounds b_{SCC} of all SCCs $SCC \in SCCsOnLevel$ to a maximum expression. Algorithm 2 sums up the bounds of all levels for obtaining an overall bound.

Algorithm 3 computes the bound of a strongly-connected program P. First Alg. 3 checks if $P = (L, E)$ is trivial, i.e., $E = \emptyset$, and returns 1, if this is the case. Next Alg. 3 collects all norms in the set *NonIncr* that either decrease or stay equal on all transitions. Subsequently Alg. 3 checks for every norm $n \in NonIncr$ and transition $l_1 \xrightarrow{\rho} l_2 \in E$, if n is bounded from below by zero and decreases on ρ. If this is the case, Alg. 3 adds $\max(n, 0)$ to the set *DecrBnded* and $l_1 \xrightarrow{\rho} l_2$ to *BndedEdgs*. Note that the transitions included in the set *BndedEdgs* can only be executed as long as their associated norms are greater than zero. Every transition in *BndedEdgs* decreases an expression in *DecrBnded* when it is taken.

```
Procedure: BndSCC($P$)
Input: strongly-connected program $P = (L, E)$
Output: a bound $b$ on the length of the traces of $P$
if $E = \emptyset$ then return 1;
$NonIncr := \emptyset$; $DecrBnded := \emptyset$; $BndedEdgs := \emptyset$;
foreach $n \in N$ do
    if $\forall\, l_1 \xrightarrow{\rho} l_2 \in E\ n \geq n' \in \alpha(\rho)$ then
        $NonIncr := NonIncr \cup \{n\}$;
foreach $l_1 \xrightarrow{\rho} l_2 \in E,\ n \in NonIncr$ do
    if $n \geq 0, n > n' \in \alpha(\rho)$ then
        $DecrBnded := DecrBnded \cup \{\max(n, 0)\}$;
        $BndedEdgs := BndedEdgs \cup \{l_1 \xrightarrow{\rho} l_2\}$;
if $BndedEdgs = \emptyset$ then fail with "there is no bound for $P$";
$b = $ Bound($(L, E \setminus BndedEdgs)$);
return $((\sum DecrBnded) + 1) \cdot b$;
```

Algorithm 3. BndSCC computes a bound for a single SCC

As the expressions in *DecrBnded* are never increased, the sum of all expressions in *DecrBnded* is a bound on how often the transitions in *BndedEdgs* can be taken. If *DecrBnded* is empty, Alg. 2 fails, because the absence of infinite cycles could not be proven. Otherwise we recursively call Alg. 2 on $(L, E \setminus BndedEdgs)$ for a bound b on this subgraph. The subgraph can at most be entered as often as the transitions in *BndedEdgs* can be taken plus one (when it is entered first). Thus, $((\sum DecrBnded) + 1) \cdot b$ is an upper bound for P.

Role of SCA in our Bound Analysis. Our bound analysis uses the size-change abstractions of transitions to determine how a norm n changes according to $n \geq n'$, $n > n'$, $n \geq 0$ in Alg. 3. We plan to incorporate inequalities between different norms (like $n \geq m'$) in future work to make our analysis more precise.

Termination analysis. If in Algorithm 2 the global invariant analysis cannot infer an upper bound on some local bound, the algorithm fails to compute a bound, but we can still compute a lexicographic ranking function, which is sufficient to prove termination. The respective adjustment of our algorithm is straightforward.

We give an example for the application of Algorithm 2 to Example 2 and to the flagship example of [13] in the extended version of this paper.

7 Heuristics for Extracting Norms

In this section we describe our heuristic for extracting norms from programs. Let $P = (L, E)$ be a program and $l \in L$ be a location. We compute all cycle-free paths from l back to l. For all arithmetic conditions $x \geq y$ appearing in some of these paths we take $x - y$ as a norm if $x - y$ decreases on this path; this can be checked by an SMT solver. Note that in such a case $x - y$ is a local ranking function for this program path. Similar patterns and checks can be implemented for iterations over bitvectors and data structures. For a more detailed discussion on how to extract the local ranking functions of a program path we refer the reader to [15]. We also compute norms for inner loops on which already extracted norms are control dependent and add them to the set of norms until a fixed point is reached (similar to program slicing). We also include the sum and the difference of two norms, if an inner loop affects two norms at the same time. Further, we include the rounded logarithm of a norm, if the norm is multiplied by a constant on some program path. In general any integer-valued expression can be used as a program norm, if considered useful by some heuristic. Clearly the analysis gets more precise the more norms are taken into account, but also more costly.

8 Related Work

In recent work [16] it was shown that SCA [20,3] is an instance of the more general technique of transition predicate abstraction (TPA) [24]. We argue that precisely because of its limited expressiveness SCA is suitable for bound analysis: abstracted programs are simple enough such that we can compute bounds for them. While we describe how to make imperative programs amenable to bound analysis by SCA, [16] is not concerned with practical issues. The Terminator tool [7] uses TPA for constructing a Ramsey based termination argument [23]. This argument does not allow to construct ranking functions or bounds from termination proofs and requires reasoning about the transitive hulls of programs by a software model checker, which is the most expensive step in the analysis. Our light-weight static analysis uses SCA for composing global bounds from bounds on norms and requires only the computation of transitive hulls of abstracted programs, which is markedly less expensive. Recent papers [18,26] propose to construct abstract programs for which termination is obvious and which are closed under transitivity in order to avoid analyzing transitive hulls of programs. Others [15,4] propose to lift standard abstract domains (as octagon, polyhedra) to powerset domains for the computation of disjunctive transition invariants, which requires a difficult lifting of the widening operator. In contrast, SCA is a finite powerset abstract domain, which can handle disjunction but does not require widening. In earlier work [15] we stated ad hoc proof rules for computing global bounds from local bounds of transitions. We generalize these rules by our bound algorithm and identify SCA as a suitable abstraction for bound analysis. Our contextualization technique is a sound generalization of the flawed *enabledness check* of [15]. Like our paper, [13] proposes the use of program transformation for bound analysis. While the algorithm of [13] is involved and parameterized by an abstract domain, our program transformations are easy to implement and rely only on an SMT solver. Despite its success in functional/declarative languages, e.g. [21,17], SCA [20,3] has not yet been applied to imperative programs. We are the first to describe such an analysis.

We give a detailed comparison with the cited approaches and others in the extended version.

9 Experiments

Our tool Loopus applies the methods of this paper to obtain upper bounds on loop iterations. The tool employs the LLVM compiler framework and performs its analysis on the LLVM intermediate representation [19]. We are using ideal integers in our analysis instead of exact machine representation (bitvectors). Our analysis operates on the SSA variables generated by the mem2reg pass and handles memory references using optimistic assumptions. For logical reasoning we use the *yices* SMT solver [8]. Our experiments were performed on an Intel Xeon CPU (4 cores with 2.33 GHz) with 16 GB Ram. Loopus computed a bound for 93% of the 262 loops of the Mälardalen WCET [2] benchmark (72% for loops with more than one path) in less than 35 seconds total time. Loopus computed a bound for 75% (65% for loops with more than one path) of the 4090 loops of the compiler optimization benchmark cBench [1] within a 1000 seconds timeout for each loop (3923 loops in less than 4 seconds). 1102 of the 4090 loops required the summarization of inner loops. Loopus computed a bound for 56% of these loops. We give more details about our experiments and the cases in which we failed in the extended version of this paper.

Acknowledgement. We would like to thank the anonymous reviewers for their insightful comments.

References

1. http://ctuning.org/wiki/index.php/CTools:CBench
2. http://www.mrtc.mdh.se/projects/wcet/benchmarks.html
3. Ben-Amram, A.M.: Monotonicity constraints for termination in the integer domain. Technical report (2011)
4. Berdine, J., Chawdhary, A., Cook, B., Distefano, D., O'Hearn, P.W.: Variance analyses from invariance analyses. In: POPL, pp. 211–224 (2007)
5. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: FMCAD, pp. 25–32 (2009)
6. Colby, C., Lee, P.: Trace-based program analysis. In: POPL, pp. 195–207 (1996)
7. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: PLDI, pp. 415–426 (2006)
8. Dutertre, B., de Moura, L.: The yices smt solver. Technical report (2006)
9. Goldsmith, S., Aiken, A., Wilkerson, D.S.: Measuring empirical computational complexity. In: ESEC/SIGSOFT FSE, pp. 395–404 (2007)
10. Gopan, D., Reps, T.W.: Lookahead widening. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 452–466. Springer, Heidelberg (2006)
11. Gulavani, B.S., Gulwani, S.: A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 370–384. Springer, Heidelberg (2008)
12. Gulwani, S.: SPEED: Symbolic complexity bound analysis. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 51–62. Springer, Heidelberg (2009)

13. Gulwani, S., Jain, S., Koskinen, E.: Control-flow refinement and progress invariants for bound analysis. In: PLDI, pp. 375–385 (2009)
14. Gulwani, S., Mehra, K.K., Chilimbi, T.M.: Speed: precise and efficient static estimation of program computational complexity. In: POPL, pp. 127–139 (2009)
15. Gulwani, S., Zuleger, F.: The reachability-bound problem. In: PLDI, pp. 292–304 (2010)
16. Heizmann, M., Jones, N.D., Podelski, A.: Size-change termination and transition invariants. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 22–50. Springer, Heidelberg (2010)
17. Krauss, A.: Certified size-change termination. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 460–475. Springer, Heidelberg (2007)
18. Kroening, D., Sharygina, N., Tsitovich, A., Wintersteiger, C.M.: Termination analysis with compositional transition invariants. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 89–103. Springer, Heidelberg (2010)
19. Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis & transformation. In: CGO 2004: Proceedings of the International Symposium on Code Generation and Optimization, p. 75. IEEE Computer Society, Washington, DC, USA (2004)
20. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: POPL, pp. 81–92 (2001)
21. Manolios, P., Vroon, D.: Termination analysis with calling context graphs. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 401–414. Springer, Heidelberg (2006)
22. Monniaux, D.: Automatic modular abstractions for linear constraints. In: POPL, pp. 140–151 (2009)
23. Podelski, A., Rybalchenko, A.: Transition invariants. In: LICS, pp. 32–41 (2004)
24. Podelski, A., Rybalchenko, A.: Transition predicate abstraction and fair termination. In: POPL, pp. 132–144 (2005)
25. Popeea, C., Chin, W.-N.: Inferring disjunctive postconditions. In: Okada, M., Satoh, I. (eds.) ASIAN 2006. LNCS, vol. 4435, pp. 331–345. Springer, Heidelberg (2008)
26. Tsitovich, A., Sharygina, N., Wintersteiger, C.M., Kroening, D.: Loop summarization and termination analysis. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 81–95. Springer, Heidelberg (2011)

Satisfiability Modulo Recursive Programs

Philippe Suter*, Ali Sinan Köksal, and Viktor Kuncak

École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
{firstname.lastname}@epfl.ch

Abstract. We present a semi-decision procedure for checking satisfiability of expressive correctness properties of recursive first-order functional programs. In our approach, both properties and programs are expressed in the same language, a subset of Scala. We implemented our procedure and integrated it with the Z3 SMT solver and the Scala compiler. Our procedure is sound for counterexamples and for proofs of terminating functions. It is terminating and thus complete for many important classes of specifications, including all satisfiable formulas and all formulas where recursive functions satisfy certain syntactic restrictions. Using our system, Leon, we verified detailed correctness properties for functional data structure implementations, as well as syntax tree manipulations. We have found our system to be fast for both finding counterexamples and finding correctness proofs, and to scale to larger programs than alternative techniques.

1 Introduction

This paper explores the problem of reasoning about functional programs. We reduce this problem to solving constraints representing precisely program semantics. Our starting point are SMT solving tools [3,10,22], which proved to be important drivers of advances in verification of software and hardware [2,12]. SMT solvers are efficient for deciding quantifier-free formulas in the language of useful theories, such as linear integer arithmetic, algebraic data types, and finite sets. Nonetheless, the operations available in the existing theories are limited, which prevents verification against more detailed specifications, as well as automated discovery of more complex invariants needed for verification. To increase the power of SMT-based reasoning, we extend the expressive power of formulas and allow them to contain user-defined recursive functions. By insisting on computable (as opposed to arbitrarily axiomatizable) functions, we obtain *familiarity* to developers, as well as *efficiency* and *completeness* of reasoning.

Our technique extends SMT solvers with recursive function definitions, so it can be used for all tasks where SMT solvers are used, including verification of functional and imperative programs, synthesis, and test generation. In this paper we introduce this technique and its concrete implementation as a verifier, named Leon, for a functional subset of Scala [25]. Leon enables the developer to state the properties as pure Scala functions and compile them using the standard Scala

* Philippe Suter was supported by the Swiss NSF Grant 200021_120433.

E. Yahav (Ed.): SAS 2011, LNCS 6887, pp. 298–315, 2011.

compiler. Developers never leave the familiar world of Scala; they use the existing library for dynamically checking executable contracts [24] to describe the desired properties. As a result, run-time contract violations can be found using testing, which would be difficult if there was a significant gap between annotation and implementation language [32]. Because we can use other functions to specify properties of the functions of interest, we obtain a very expressive specification language. We can use abstraction functions to specify abstract views of data structures. We can naturally specify properties such as commutativity and idempotence of operations, which require multiple function invocations and are not easy to express in type systems and many other specification approaches.

Leon generates verification conditions that enforce 1) that the functions meet their contracts, 2) that the preconditions at all function invocations are met, and that 3) the pattern-matching is complete for given preconditions. Note that, to define an external property of a set of functions, the developer can write a boolean-valued test function that invokes the functions of interest, and state a contract that the function always returns **true**. Leon searches in parallel for proofs and counterexamples for all generated verification conditions.

Contributions. The core technical result of our paper is an algorithm for combined proof and counterexample search, as well as a theoretical and experimental analysis of its effectiveness. We summarize our contributions as follows:

- We introduce a procedure for satisfiability modulo computable functions, which integrates DPLL(T) solving with unfolding of function definitions and validation of candidate models, which returns a model or UNSAT.
- We establish that our satisfiability procedure is:
  1. sound for models: every model it returns makes the formula true;
  2. terminating for all formulas that are satisfiable;
  3. sound for proofs: if it reports UNSAT, then there are no models;
  4. terminating for all sufficiently surjective abstractions [28];
  5. satisfying the above properties if the declared functions are always terminating; more generally, UNSAT implies no "terminating models", moreover, each returned model leads to **true**.
- We describe the implementation of our system, named Leon, as a plugin for the Scala compiler, which uses only existing constructs in Scala for specifying functions and properties. The system integrates with the SMT solver Z3.
- We present our results in verifying over 60 functions manipulating integers, sets, and algebraic data types, with detailed invariants of complex data structures such as red-black trees and amortized heap, and user code such as syntax tree manipulation. Leon verified detailed correctness properties about the content of data as well as completeness of pattern-matching expressions.

We have found that Leon was fast for finding both counterexamples and proofs for verification conditions. We thus believe that the algorithm holds great promise for practical verification of complex properties of computer systems. Leon and all benchmarks are available from `http://lara.epfl.ch`.

2 Examples

We now illustrate how Leon can be used to prove interesting properties about functional programs. Consider the following recursive datatype, written in Scala syntax [25], that represent formulas of propositional logic:

```
sealed abstract class Formula
case class And(lhs: Formula, rhs: Formula) extends Formula
case class Or(lhs: Formula, rhs: Formula) extends Formula
case class Implies(lhs: Formula, rhs: Formula) extends Formula
case class Not(f: Formula) extends Formula
case class PropVar(id: Int) extends Formula
```

We can write a recursive function that simplifies a formula by rewriting implications into disjunctions as follows:

```
def simplify(f: Formula): Formula = (f match {
  case And(lhs, rhs) ⇒ And(simplify(lhs), simplify(rhs))
  case Or(lhs, rhs) ⇒ Or(simplify(lhs), simplify(rhs))
  case Implies(lhs, rhs) ⇒ Or(Not(simplify(lhs)), simplify(rhs)) // note the replacement
  case Not(f) ⇒ Not(simplify(f))
  case PropVar(_) ⇒ f
}) ensuring(isSimplified(_))
```

Note that **ensuring** is an infix command in Scala, taking the entire function body as the left argument and taking a lambda function as the right argument. The expression e **ensuring** p indicates that p(e) should hold for all values of the function parameters. To denote an anonymous function, $\lambda x.B$, in Scala we write x⇒B. When there is only one occurrence of x in B, we can denote this occurrence by _ and omit the binder, so both (_+1) and x⇒x+1 denote the increment function.

We can write an executable function isSimplified that checks whether a given formula contains an implication as a subformula, and use it in a contract. The **ensuring** statement in the example is a postcondition written in Scala notation [24], stating that the function isSimplified evaluates to true on the result. We define isSimplified recursively as follows:

```
def isSimplified(f: Formula): Boolean = f match {
  case And(lhs, rhs) ⇒ isSimplified(lhs) && isSimplified(rhs)
  case Or(lhs, rhs) ⇒ isSimplified(lhs) && isSimplified(rhs)
  case Implies(_,_) ⇒ false
  case Not(f) ⇒ isSimplified(f)
  case PropVar(_) ⇒ true
}
```

Note that a developer would also typically write such an executable specification function for testing purposes. Using our procedure for satisfiability modulo computable functions, Leon can prove that the postcondition of simplify is satisfied for *every* input formula f.

Such subsets of values denoted by algebraic data types are known as refinement types [13]. Refinement types that are defined using functions such as isSimplified are in fact sufficiently surjective abstractions [28], which implies that our system is a decision procedure for such constraints (see Section 3.1). This is confirmed with our experiments—our tool is predictably fast on such examples.

Suppose now that we wish to prove that simplifying a simplified formula does not change it further. In other words, we want to prove that the property simplify(simplify(f)) == simplify(f) holds for all formulas f. Because our programming and specification languages are identical, we can write such universally quantified statements as functions that return a boolean and whose postcondition is that they always return true. In this case, we would write:

```
def simplifyIsStable(f: Formula) : Boolean = {simplify(simplify(f)) == simplify(f)} holds
```

Because such specifications are common, we use the notation **holds** instead of the more verbose postcondition stating that the returned result should be an identity function with boolean argument **ensuring**(res⇒res). Our verification system proves this property instantly.

Another application for our technique is verifying that pattern-matching expressions are defined for all cases. Pattern-matching is a very powerful construct commonly found in functional programming languages. Typically, evaluating a pattern-matching expression on a value not covered by any case raises a runtime error. Because checking that a **match** expression never fails is difficult in non-trivial cases (for instance, in the presence of guards), compilers in general cannot statically enforce this property. For instance, consider the following function that computes the set of variables in a propositional logic formula, assuming that the formula has been simplified:

```
def vars(f: Formula): Set[Int] = {
  require(isSimplified(f))
  f match {
    case And(lhs, rhs) ⇒ vars(lhs) ++ vars(rhs)
    case Or(lhs, rhs) ⇒ vars(lhs) ++ vars(rhs)
    case Not(f) ⇒ vars(f)
    case PropVar(i) ⇒ Set[Int](i) }}
```

Here ++ denotes the set union operation in Scala. Although it is implied by the precondition that all cases are covered, the Scala compiler on this example will issue the warning:

```
Logic.scala: warning: match is not exhaustive!
missing combination        Implies
```

Previously, researchers have developed specialized analyses for checking such exhaustiveness properties [9,11]. Our system generates verification conditions for checking the exhaustiveness of all pattern-matching expressions, and then uses the same procedure to prove or disprove them as for the other verification conditions. It quickly proves that this particular example is exhaustive by unrolling the definition of isSimplified sufficiently many times to conclude that t can never be an Implies term. Note that our system will also prove that all recursive calls to vars satisfy its precondition; it performs sound assume-guarantee reasoning.

Consider now the following function, that supposedly computes a variation of the negation normal form of a formula f:

```
def nnf(formula: Formula): Formula = formula match {
  case Not(Not(f)) ⇒ nnf(f)
  case And(lhs, rhs) ⇒ And(nnf(lhs), nnf(rhs))
  case Not(And(lhs, rhs)) ⇒ Or(nnf(Not(lhs)), nnf(Not(rhs)))
  ...
  case Implies(lhs, rhs) ⇒ Implies(nnf(lhs), nnf(rhs))
}
```

From the supposed roles of the functions simplify and nnf, one could conjecture that the operations are commutative. Because of the treatment of implications in the above definition of nnf, though, this is not the case. We can disprove this property by finding a counterexample to

```
def wrongCommutative(f: Formula) : Boolean = {
  nnf(simplify(f)) == simplify(nnf(f))} holds
```

On this input, Leon reports

```
Error: Counter-example found:
f -> Implies(Not(And(PropVar(48), PropVar(47))),
                And(PropVar(46), PropVar(45)))
```

A consequence of our algorithm is that Leon never reports false positives (see Section 3.1). In this particular case, the counterexample clearly shows that there is a problem with the treatment of implications whose left-hand side contains a negation. Counterexamples such as this one are typically short and Leon finds them quickly.

As a final example of the expressive power of our system, we consider the question of showing that an implementation of a collection implements the proper interface. Consider the implementation of a set as red-black trees. (We omit the datatype definition in the interest of space.) To specify the operation on the trees in terms of the set interface that they are supposed to implement, we define an *abstraction function* that computes from a tree the set it represents:

```
def content(t : Tree) : Set[Int] = t match {
  case Empty() ⇒ Set.empty
  case Node(_, l, v, r) ⇒ content(l) ++ Set(v) ++ content(r) }
```

Note that this is again a function one would write for testing purposes. The specification of insertion using this abstraction becomes very natural, despite the relative complexity of the operations:

```
def ins(x: Int, t: Tree): Tree = (t match {
    case Empty() ⇒ Node(Red(),Empty(),x,Empty())
    case Node(c,a,y,b) ⇒ if (x < y) balance(c, ins(x, a), y, b)
                         else if (x == y) Node(c,a,y,b)
                         else balance(c,a,y,ins(x, b)) }
}) ensuring (res⇒ content(res) == content(t) ++ Set(x))
```

We also wrote functions that check whether a tree is balanced and whether it satisfies the coloring properties. We used these checks to specify insertion and balancing operations. Leon proved all these properties of red-black tree operations. We present more such results in Section 5.

3 Our Satisfiability Procedure

In this section, we describe our algorithm for checking the satisfiability of formulas modulo recursive functions. We start with a description of the supported class of formulas. Let $\mathcal{L}$ be a *base theory* (logic) with the following properties:

- $\mathcal{L}$ is closed under propositional combination and supports boolean variables
- $\mathcal{L}$ supports uninterpreted function symbols
- there exists a complete decision procedure for $\mathcal{L}$

Note that the logics supported by DPLL(T) SMT solvers naturally have these properties.[1]

Let $\mathcal{L}^{\Pi}$ be the extension of $\mathcal{L}$ with *interpreted* function symbols defined in a program Π. The interpretation is given by an expression in $\mathcal{L}^{\Pi}$ (the *implementation*). To facilitate proofs and the description of program invariants, functions in Π can also be annotated with a *pre-* and *postcondition*. We denote the implementation, pre- and postcondition of a function f in Π by impl_f^{Π}, prec_f^{Π} and post_f^{Π} respectively. The free variables of these expressions are the arguments of f denoted args_f^{Π}, as well as, for the postcondition, a special variable ρ that denotes the result of the computation.

```
def solve($\phi$, $\Pi$) {
  ($\phi$, $B$) = unrollStep($\phi$, $\Pi$, $\emptyset$)
  while(true) {
    decide($\phi \wedge \bigwedge_{b \in B} b$) match {
      case "SAT" ⇒ return "SAT"
      case "UNSAT" ⇒ decide($\phi$) match {
        case "UNSAT" ⇒ return "UNSAT"
        case "SAT" ⇒ ($\phi$, $B$) = unrollStep($\phi$, $\Pi$, $B$) }}}}
```

Fig. 1. Pseudo-code of the solving algorithm. The decision procedure for the base theory is invoked through the calls to decide.

Figure 1 shows the pseudo-code of our algorithm. It is defined in terms of two subroutines, decide, which invokes the decision procedure for $\mathcal{L}$, and unrollStep, whose description follows. Note that the algorithm maintains, along with a formula ϕ, a set B of boolean literals. We call these *control literals* and their role is described below.

At a high-level, the role of unrollStep is to give a partial interpretation to function invocations, which are treated as uninterpreted in $\mathcal{L}$. This is achieved in two steps: 1) introduce definitions for one unfolding of the (uninterpreted) function invocations in ϕ and 2) generate an assignment of control literals that guard newly introduced function invocations. As an example, consider a formula ϕ that contains the term size(lst), where lst is a list and size is the usual recursive

[1] In Leon, $\mathcal{L}$ is the multi-sorted combination of uninterpreted function symbols with integer linear arithmetic and user-defined algebraic datatypes and finite sets.

```
size(lst) = lst match {
  case Nil ⇒ 0
  case Cons(_, xs) ⇒ 1 + size(xs)
}
```

$$
\begin{aligned}
\psi \equiv\ & \mathsf{size}(\mathsf{lst}) = t_{\mathsf{f}} \\
& \wedge b_{\mathsf{f}} \Leftrightarrow \mathsf{lst} = \mathsf{Nil} \\
& \wedge b_{\mathsf{f}} \Rightarrow t_{\mathsf{f}} = 0 \\
& \wedge \neg b_{\mathsf{f}} \Rightarrow t_{\mathsf{f}} = 1 + \mathsf{size}(\mathsf{lst.tail})
\end{aligned}
$$

Fig. 2. Function definition and its translation into clauses with control literals

definition of its length. Figure 2 shows on the left the definition of size(lst) and on the right its encoding into clauses with fresh variables.

This encoding into clauses is obtained by recursively introducing, for each if-then-else term, a boolean variable representing the truth value of the branching condition, and another variable representing the value of the if-then-else term.

In addition to conjoining ψ to ϕ, unrollStep would produce the set of literals $\{b_{\mathsf{f}}\}$. The set should be understood as follows: the decision procedure for the base theory $\mathcal{L}$, which treats size as an uninterpreted function symbol, if it reports SAT, can only be trusted when b_{f} is set to true. Indeed, if b_{f} is false, the value used for t_{f} and hence for the term size(lst) may depend on size(lst.tail), which is undefined (because its definition has not yet been introduced). A subsequent call to unrollStep on $\phi \wedge \psi$ would introduce the definition of size(lst.tail). When unrolled functions have a precondition, the definitions introduced for their body and postcondition are guarded by the precondition. This is done to prevent an inconsistent function definition with a precondition equivalent to **false** from making the formula unsatisfiable.

The formula in $\mathcal{L}$ without the control literals can be seen as an *under-approximation* of the formula with the semantics of the program defining $\mathcal{L}^{\Pi}$, in that it accepts all the same models plus some models in which the interpretation of some invocations is incorrect, and the formula with the control literals is an *over-approximation*, in the sense that it accepts only the models that do not rely on the guarded invocations. This explains why the UNSAT answer can be trusted in the first case and the SAT case in the latter.

In Figure 1, the third argument of calls to unrollStep denotes the set of control literals introduced at previous steps. An invocation of unrollStep will insert definitions for *all* or only *some* function terms that are guarded by such control literals, and the returned set will contain all literals that were not released as well as the newly introduced ones. From an abstract point of view, when a new control literal is created, it is inserted in a global priority queue with all the function invocations that it guards. An important requirement is that the dequeuing must be *fair*: any control literal that is enqueued must eventually be dequeued. This fairness condition guarantees that our algorithm is complete for satisfiable formulas (see Section 3.1). Using a first-in first-out policy, for instance, is enough to guarantee fairness and thus completeness. Finally, note that unrollStep not only adds definitions for the implementation of the function calls, but also for their postcondition when it exists. We discuss the issue of reliably using these facts in Section 3.1.

Implementation notes. While the description of solve suggests that we need to query the solver twice in each loop iteration, we can in practice use the solver's capability to output *unsat cores* to detect with a single query whether the conjunction of control literals $\bigwedge_{b \in B} b$ played any role in the unsatisfiability. Similarly, when adding the new constraints obtained from unrollStep, we can use the solver's incremental reasoning and push the new constraints directly, rather than building a new formula and issuing a new query. SMT solvers can thus exploit at any time facts learned in previous iterations.

Finally, although we noted that we cannot in general trust the underlying solver when it reports SAT for the formula ϕ without control literals, it could still be that the assignment the solver guessed for the uninterpreted function symbols is valid. Because testing an assignment is fast (it amounts to executing the specification), we can therefore sometimes report SAT early and save time.

3.1 Properties of Our Procedure

The properties of our procedure rely on the following two assumptions.

Termination: All precondition computations terminate for all values. Each function in the program Π terminates on each input for which the precondition holds, and similarly for each postcondition. Tools such as [14,1] or techniques developed for ACL2 [20] could be used to establish this property.

Base theory solver soundness: The underlying solver is complete and sound for the (quantifier-free) formulas in the base theory. The completeness means that each model that the solver reports should be a model for the conjunction of all constraints passed to the solver. Similarly, soundness means that whenever the solver reports unsatisfiability, false can be logically concluded modulo the solver's theories from these constraints.

We use the above assumptions throughout this section. Note, however, that even without the termination assumption, a counterexample reported by Leon is never a counterexample that generates a terminating execution of the property resulting in the value true, so it is a counterexample worth inspecting.

Soundness for Proofs. Our algorithm reports unsatisfiability if and only if the underlying solver could prove unsatisfiable the problem given to it *without* the control literals. Because the control literals are not present, some function applications are left uninterpreted, and the conclusion that the problem is unsatisfiable therefore applies to *any* interpretation of the remaining function application terms, and in particular to the one conforming to the correct semantics.

From the assumption that the underlying solver only produces sound proofs, it suffices to show that all the axioms communicated to the solver as a result of the unrollStep invocations are obtained from sound derivations. These are correct by definition: they are logical consequences obtained by the definition of functions, and these definitions are conservative when the functions are terminating.

An important consideration when discussing soundness of the *post* axioms is that any proof obtained with our procedure can be considered valid only when the following properties about the functions of Π have been proved:[2]

1. for each function f of Π, the following formula must hold:

$$\mathsf{prec}_f^{\Pi} \implies \mathsf{post}_f^{\Pi}\left[\mathsf{impl}_f^{\Pi}/\rho\right]$$

2. for each call in f to a function f_2 (possibly f itself), the precondition $\mathsf{prec}_{f_2}^{\Pi}$ must be implied by the path condition
3. for each pattern-matching expression, the patterns must be shown to cover all possible inputs under the path condition.

The above conditions guarantee the absence of runtime errors, and they also allow us to prove the overall correctness by induction on the call stack, as is standard in assume-guarantee reasoning for sequential procedures without side effects [15, Chapter 12].

The first condition shows that all postconditions are logical implications of the function implementations under the assumption that the preconditions hold. The second condition shows that all functions are called with arguments satisfying the preconditions. Because all functions terminate, it follows that we can safely assume that postconditions always hold for all function calls. This justifies the soundness of axioms *post* in the presence of ϕ and Π.

Soundness for Models. Our algorithm reports satisfiability when the solver reports that the unrolled problem augmented with the control literals is satisfiable. By construction of the set of control literals, it follows that the solver can only have used values for function invocations whose definition it knows. As a consequence, every model reported by the solver for the problem augmented with the control literals is always a true model of the original formula. We mentioned in Section 3 that we can also check other satisfying assignments produced by the solver. In this second case, we use an evaluator that complies with the semantics of the program, and therefore the validated models are true models as well.

Termination for Satisfiable Formulas. Our procedure has the remarkable property that it finds a model whenever the model for a formula exists. We define a model as an assignment to the free variables of the formula such that evaluating it under that assignment terminates with the value **true**. An assignment that leads to a diverging evaluation is not considered to be a proper model. To understand why our procedure always finds models when they exist, consider a counterexample for the specification. This counterexample is an assignment of integers and algebraic data types to variables of a function $f(\boldsymbol{x})$ being proved.

[2] When proving or disproving a formula ϕ modulo the functions of Π, it is in fact sufficient that the three properties hold only for all functions in ϕ and those that can be called (transitively) from them or from their contracts.

This evaluation specifies concrete inputs $\boldsymbol{a}$ for f such that evaluating $f(\boldsymbol{a})$ yields a value for which the postcondition of f evaluates to false (the case of preconditions or pattern-matching is analogous). Consider the computation tree arising from (call-by-value) evaluation of f and its postcondition. This computation tree is finite. Consequently, the tree contains finitely many unrollings of function invocations. Let us call K the maximum depth of that tree. Consider now the execution of the algorithm in Figure 1; because we assume that any function invocation that is guarded by a control literal is eventually accessible, we can safely conclude that every function application in the original formula will eventually be unrolled. By applying this reasoning inductively, we conclude that eventually, all function applications up to nesting level $K + 1$ will be unrolled. This means that the computation tree corresponding to $f(\boldsymbol{a})$ has also been explored. By the completeness of the solver for the base theory and the consistency of a satisfying specification, it means that the solver reports a counterexample (either $\boldsymbol{a}$ itself or another counterexample).

Termination for Sufficiently Surjective Abstraction. Our procedure always terminates and is therefore a decision procedure in the case of a recursive function that are *sufficiently surjective catamoprhisms* [28]. In fact, it also serves as the first implementation of the technique [28]. A catamorphism is a fold function from a tree data type to some domain, which uses a simple recursive pattern: compute the value on subtrees, then combine these values using an expressions from a decidable logic. The sufficient surjectivity for a function f is a condition implying, informally, that the size of the set $\{x \mid f(x) = y\}$ can be made sufficiently large for "sufficiently large" elements y. Leon shows that the technique inspired by [28] is fast in practice. Moreover, by interleaving unrolling and satisfiability checking, it addresses in practice the open problem of determining the maximal amount of unrolling needed for a user-defined function.

We have already established termination in the case of formula satisfiability. In the case of an unsatisfiable formula, the termination follows because the unsatisfiability can be detected by unrolling the function definitions a finite number of times [28]. The unrolling depth depends on the particular sufficiently surjective abstraction, which is why [28] presents only a family of decision procedures and not a decision procedure for all sufficiently surjective abstractions. In contrast, our approach is one uniform procedure that behaves as a decision procedure for the *entire* family, because it unrolls functions in a fair way. [3]

Among the examples of such recursive functions for which our procedure is a decision procedure are functions of algebraic data types such as size, height, or content (expressed as a set, multiset, or a list). Further examples include

[3] Using the terminology of [28, p.207], consider the case of a sufficiently surjective catamorphism with the associated parametric formula M_p and set of shapes S_p, and an unsatisfiable formula containing a term $\alpha(t)$. Sufficiently unrolling α will result in coverage of all possible shapes of t that belong to S_p. If no satisfying assignment for t can be found with these shapes, then a formula at least as strong as $M_p(\alpha(t))$ will be implied, so a complete solver for the base theory will thus detect unsatisfiability because [28] detects unsatisfiability when using $M_p(\alpha(t))$.

properties such as sortedness of a list or a tree, or a combination of any finite number of functions into a finite domain. Through experiments with Leon, we have also discovered a new and very useful instance of constraints for which our tool is complete: the predicates expressing refinement types [13], which we specify as, e.g., the isSimplified function in Section 2. These functions map data structures into a finite domain–booleans, so they are sufficiently surjective. This explains why Leon is complete, and why it was so successful in verifying complex pattern-matching exhaustiveness constraints on syntax tree transformations.

Non-terminating Functions. We conclude this section with some remarks on the behavior of our procedure in the presence of functions that do not terminate on all their inputs. We are interested in showing that if for an input formula the procedure returns UNSAT, then there are indeed no models whose evaluation terminates with **true**. (The property that all models are true models is not affected by non-terminating functions.) Note that it may still be the case that the procedure returns UNSAT when there is an input for which the evaluation doesn't terminate.

To see why the property doesn't immediately follow from the all-terminating case, consider the definition: **def** $\mathsf{f}(\mathsf{x} : \mathsf{Int}) = \mathsf{f}(\mathsf{x}) + 1$. Unrolling that function could introduce a contradiction $\mathsf{f}(\mathsf{x}) = \mathsf{f}(\mathsf{x}) + 1$ and make the formula immediately unsatisfiable, thus potentially masking a true satisfying assignment. However, because all introduced definitions are guarded by a control literal, the contradiction will only prevent those literals from being true that correspond to executions leading to the infinite loop.

4 The Leon Verification System

We now present some of the characteristics of the implementation of Leon, our verification system that has at its core an implementation of the procedure presented in the previous sections. Leon takes as an input a program written in a purely functional subset of Scala and produces verification conditions for all specified postconditions, calls to functions with preconditions, and match expressions in the program.

Front-end. We wrote a plugin for the official Scala compiler to use as the front-end of Leon. The immediate advantage of this approach is that all programs are parsed and type-checked before they are passed to Leon. This also allows users to write expressive programs concisely, thanks to type-inference and the flexible syntax of Scala. The subset we support allows for definitions of recursive datatypes, as shown in examples throughout this paper, as well as arbitrarily complex pattern-matching expressions over such types. The other admitted types are integers and sets, which we found to be particularly useful for specifying properties with respect to an abstract interface. In our function definitions, we allow only immutable variables for simplicity (**val**s and no **var**s in Scala terminology).

Conversion of pattern-matching. We transform all pattern-matching expressions into equivalent expressions built with if-then-else terms. For this purpose, we use predicates that test whether their argument is of a given subtype (this is equivalent to the method .isInstanceOf[T] in Scala). The translation is relatively straightforward, and preserves the semantics of pattern-matching. In particular, it preserves the property that cases are tested in their definition order. To encode the error that can be triggered if no case matches the value, we return for the default case a fresh, uninterpreted value. This value is therefore guarded by the conjunction of the negation of all matching predicates. Recall that we separately prove that all match expressions are exhaustive. When these proofs succeed, we effectively rule out the possibility that the unconstrained error value affects the semantics of the expression.

Proofs by induction. To simplify the statement and proof of some inductive properties, we defined an annotation @induct, that indicates to Leon that it should attempt to prove a property by induction on the arguments. This works only when proving a property over a variable that is of a recursive type; in these cases, we decompose the proof that the postcondition is always satisfied into subproofs for the alternatives of the datatype. For instance, when proving by induction that a property holds for all binary trees, we generate a verification condition for the case where the tree is a leaf, then for the case where it is a node, assuming that the property holds for both subtrees.

Communicating with the solver. We used Z3 [22] as the SMT solver at the core of our solving procedure. As described in Section 3, we use Z3's support for incremental reasoning to avoid solving a new problem at each iteration of our top-level loop.

Interpretation of selectors as total functions. We should note that the interpretation of selector functions in Z3 is different than in Scala, since they are considered to be total, but uninterpreted when applied to values of the wrong type. For instance, the formula $\mathsf{Nil.head} = 5$ is considered in Z3 to be satisfiable, while taking the head of an empty list has no meaning in Scala (if not a runtime error). This discrepancy does not affect the correctness of Leon, though, as the type-checking algorithm run by the Scala compiler succeeds only when it can guarantee that the selectors are applied only to properly typed terms.

5 Experimental Evaluation

We are very excited about the speed and the expressive power of properties that Leon can prove; this feeling is probably best understood by trying out the Leon distribution. As an illustration, we here report results of Leon on proving correctness properties for a number of functional programs, and discovering counterexamples when functions did not meet their specification. A summary of our evaluation can be seen in Figure 3. In this figure, LOC denotes the number of

Benchmark *(LOC)*	#p.	#m.	V/I	U	Time	function	#p.	#m.	V/I	U	Time
ListOperations *(107)*											
size	0	1	V	1	0.12	sizeTailRecAcc	1	1	V	1	0.01
sizesAreEquiv	0	0	V	2	<0.01	sizeAndContent	0	0	V	1	<0.01
reverse	0	0	V	2	0.02	reverse0	0	1	V	2	0.04
append	0	1	V	1	0.03	nilAppend	0	0	V	1	0.03
appendAssoc	0	0	V	1	0.03	sizeAppend	0	0	V	1	0.04
concat	0	0	V	1	0.04	concat0	0	2	V	2	0.29
zip	1	2	V	2	0.09	sizeTailRec	1	0	V	1	<0.01
content	0	1	V	0	<0.01						
AssociativeList *(50)*											
update	0	1	V	1	0.03	updateElem	0	2	V	1	0.05
readOverWrite	0	1	V	1	0.10	domain	0	1	V	0	0.05
find	0	1	V	1	<0.01						
InsertionSort *(99)*											
size	0	1	V	1	0.06	sortedIns	1	1	V	2	0.24
buggySortedIns	1	1	I	1	0.08	sort	1	1	V	1	0.03
contents	0	1	V	0	<0.01	isSorted	0	1	V	1	<0.01
RedBlackTree *(117)*											
ins	2	1	V	3	2.88	makeBlack	0	0	V	1	0.02
add	2	0	V	2	0.19	buggyAdd	1	0	I	3	0.26
balance	0	1	V	3	0.13	buggyBalance	0	1	I	1	0.12
content	0	1	V	0	<0.01	size	0	1	V	1	0.11
redNHaveBlckC.	0	1	V	1	<0.01	redDHaveBlckC.	0	1	V	0	<0.01
blackHeight	0	1	V	1	<0.01						
PropositionalLogic *(86)*											
simplify	0	1	V	2	0.84	nnf	0	1	V	1	0.37
wrongCommutative	0	0	I	3	0.44	simplifyBreaksNNF	0	0	I	1	0.28
nnfIsStable	0	0	V	1	0.17	simplifyIsStable	0	0	V	1	0.12
isSimplified	0	1	V	0	<0.01	isNNF	0	1	V	1	<0.01
vars	6	1	V	1	0.13						
SumAndMax *(46)*											
max	2	1	V	1	0.13	sum	0	1	V	0	<0.01
allPos	0	1	V	0	<0.01	size	0	1	V	1	<0.01
prop0	1	0	V	1	0.02	property	1	0	V	1	0.11
SearchLinkedList *(48)*											
size	0	1	V	1	0.11	contains	0	1	V	0	<0.01
firstZero	0	1	V	1	0.03	firstZeroAtPos	0	1	V	0	<0.01
goal	0	0	V	1	0.01						
AmortizedQueue *(124)*											
size	0	1	V	1	0.14	content	0	1	V	0	<0.01
asList	0	1	V	0	<0.01	concat	0	1	V	1	0.04
isAmortized	0	1	V	0	<0.01	isEmpty	0	1	V	0	<0.01
reverse	0	1	V	3	0.20	amortizedQueue	0	0	V	2	0.05
enqueue	0	1	V	1	<0.01	front	0	1	V	3	0.01
tail	0	1	V	3	0.15	propFront	1	1	V	3	0.07
enqueueAndFront	1	0	V	4	0.21	enqDeqThrice	5	0	V	5	2.48

Fig. 3. Summary of evaluation results

lines of code, #p. denotes the number of verification conditions for function invocations with preconditions, #m. denotes the number of conditions for showing exhaustiveness of pattern-matchings, V/I denotes whether the verification conditions were valid or invalid, U denotes the maximum depth for unrolling function definitions, and Time denotes the total running time in seconds to verify all conditions for a function. The benchmarks were run on a computer equipped with two Intel Core 2 processors running at 2.66 GHz and 3.2 GB of RAM, using a very recent version of Z3 at the time of running the experiments (June 12, 2011). We verified over 60 functions, with over 600 lines of compactly written code and properties that often relate multiple function invocations. This includes a red-black tree set implementation including the height invariant (which most reported benchmarks for automated systems omit); amortized queue data structures, and examples with syntax tree refinement that show Leon to be useful for checking user code, and not only for data structures.[4]

The ListOperations benchmark contains a number of common operations on lists. Leon proves, e.g., that a tail-recursive version of size is functionally equivalent to a simpler version, that append is associative, and that content, which computes the set of elements in a list, distributes over append. For association lists, Leon proves that updating a list l1 with all mappings from another list l2 yields a new associative list whose domain is the union of the domains of l1 and l2. It proves the *read-over-write* property, which states that looking up the value associated with a key gives the most recently updated value. We express this property simply as:

```
def readOverWrite(l : List, e : Pair, k : Int) : Boolean = (e match {
  case Pair(key, value) ⇒
    find(updateElem(l, e), k) == (if (k == key) Some(value) else find(l, k))
}) holds
```

Leon proves properties of insertion sort such as the fact that the output of the function sort is sorted, and that it has the same size and content as the input list. The function buggySortedIns is similar to sortedIns, and is responsible for inserting an element into an already sorted list, except that the precondition that the list should be sorted is missing. On the RedBlackTrees benchmark, Leon proves that the tree implements a set interface and that balancing preserves the "red nodes have no black children" and "every simple path from the root to a leaf has the same number of black nodes" properties as well as the contents of the tree. In addition to proving correctness, we also seeded two bugs (forgetting to paint a node black and missing a case in balancing); Leon found a concise counterexample in each case. The PropositionalLogic benchmark contains functions manipulating abstract syntax trees of boolean formulas. Leon proves that, e.g., applying a negation normal form transformation twice is equivalent to applying it once.

Further benchmarks are taken from the Verified Software Competition [30]: For example, in the AmortizedQueue benchmark Leon proves that operations on an amortized queue implemented as two lists maintains the invariant that the size of the "front" list is always larger than or equal to the size of the "rear"

[4] All benchmarks and the sources of Leon are available from http://lara.epfl.ch.

list, and that the function front implements an abstract queue interface given as a sequence.

We also point out that, apart from the parameterless @induct hint for certain functions, there are no other hint mechanisms used in Leon: the programmer simply writes the code, and boolean-valued functions that describe the desired properties (as they would do for testing purposes). We thus believe that Leon is easy and simple to use, even for programmers that are not verification experts.

6 Related Work

We next compare our approach to the most closely related techniques.

Interactive verification systems. The practicality of computable functions as an executable logic has been demonstrated through a long line of systems, notably ACL2 [18] and its predecessors. These systems have been applied to a number of industrial-scale case studies in hardware and software verification [18,21]. Recent systems based on functional programs include VeriFun [31] and AProVE [14]. Moreover, computable specifications form important parts of many case studies in proof assistants Coq [5] and Isabelle [23]. These systems support more expressive logics, with higher-order quantification, but provide facilities for defining executable functions and generating the corresponding executable code in functional programming languages [16]. When it comes to reasoning within these systems, they offer varying degrees of automation. What is common is the difficulty of predicting when a verification attempt will succeed. This is in part due to possible simplification loops associated with the rewrite rules and tactics of these provers. Moreover, for performance and user-interaction reasons, interactive proofs often fail to fully utilize aggressive case splitting that is at the heart of modern SMT solvers.

Inductive generalizations vs. counterexamples. Existing interactive systems such as ACL2 are stronger in automating induction, whereas our approach is complete for finding counterexamples. We believe that the focus on counterexamples will make our approach very appealing to programmers that are not theorem proving experts. The HMC verifier [17] and DSolve [26] can automatically discover inductive invariants, so they have more automation, but it appears that certain properties involving multiple user-defined functions are not expressible in these systems. Recent results also demonstrate inference techniques for higher-order functional programs [19,17]. These approaches hold great promise for the future, but the programs on which those systems were evaluated are smaller than our benchmarks. Leon focuses on first-order programs and is particularly effective for finding counterexamples. Our experience suggests that Leon is more scalable than the alternative systems that can deal with this expressive properties. Counterexample generation has been introduced into Isabelle through tools like Nitpick [6]. Further experimental comparisons would be desirable, but these techniques do not use theory solvers and appear slower than Leon on complex functional programs. Counterexample generation has been recently incorporated into ACL2 Sedan [7]. This techniques is tied to the sophisticated ALC2 proof mechanism and uses proof failures to find counterexamples. Although it appears very useful, it does not have our completeness guarantees.

Counterexample finding systems for imperative code. Researchers have explored the idea of iterative function and loop unfolding in a number of contexts. Among well-known tools is CBMC [8]; techniques to handle procedures include [29,4,27]. The use of imperative languages in these systems makes their design more complex and limits the flexibility of the counterexample search. Thanks to a direct encoding into SMT and the absence of side-effects, Leon can prove more easily properties that would be harder to prove using imperative semantics. As a result, we were able to automatically prove detailed functional correctness properties as opposed to only checking for errors such as null dereferences. Moreover, both [29] and [27] focus on error finding, while we were also able to prove several non-trivial properties correct, using counterexample finding to debug our code and specifications during the development.

Satisfiability modulo theory solvers. SMT solvers behave as (complete) decision procedures on certain classes of quantifier-free formulas containing theory operations and uninterpreted functions. However, they do not support user-defined functions, such as functions given by recursive definitions. An attempt to introduce them using quantifiers leads to formulas on which the prover behaves unpredictably for unsatisfiable instances, and is not able to determine whether a candidate model is a real one. This is because the prover has no way to determine whether universally quantified axioms hold for all of the infinitely many values of the domain. Leon uses terminating executable functions, whose definitions are a well-behaved and important class of quantified axioms, so it can check the consistency of a candidate assignment. A high degree of automation and performance in Leon comes in part from using state-of-the-art SMT solver Z3 [22] to reason about quantifier-free formula modulo theories, as well as to perform case splitting along with automated lemma learning. Other SMT solvers, such as CVC3 [3] could also be used.

Acknowledgments. We thank Nikolaj Bjørner and Leonardo de Moura for their help with Z3. We thank Mirco Dotta and Swen Jacobs for preliminary versions of some of the benchmarks. We thank Panagiotis Manolios and J Strother Moore for discussions about ACL2.

References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: COSTA: Design and implementation of a cost and termination analyzer for java bytecode. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2007. LNCS, vol. 5382, pp. 113–132. Springer, Heidelberg (2008)
2. Ball, T., Bounimova, E., Levin, V., Kumar, R., Lichtenberg, J.: The static driver verifier research platform. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 119–122. Springer, Heidelberg (2010)
3. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)

4. Basler, G., Kroening, D., Weissenbacher, G.: A complete bounded model checking algorithm for pushdown systems. In: Yorav, K. (ed.) HVC 2007. LNCS, vol. 4899, pp. 202–217. Springer, Heidelberg (2008)
5. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development–Coq'Art: The Calculus of Inductive Constructions. Springer, Heidelberg (2004)
6. Blanchette, J.C., Nipkow, T.: Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 131–146. Springer, Heidelberg (2010)
7. Chamarthi, H.R., Dillinger, P.C., Manolios, P., Vroon, D.: The ACL2 sedan theorem proving system. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 291–295. Springer, Heidelberg (2011)
8. Clarke, E.M., Kröning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
9. Dotta, M., Suter, P., Kuncak, V.: On static analysis for expressive pattern matching. Tech. Rep. LARA-REPORT-2008-004, EPFL (2008)
10. Dutertre, B., de Moura, L.: A Fast Linear-Arithmetic Solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
11. Ferrara, P.: Static type analysis of pattern matching by abstract interpretation. In: Hatcliff, J., Zucca, E. (eds.) FMOODS 2010. LNCS, vol. 6117, pp. 186–200. Springer, Heidelberg (2010)
12. Franzen, A., Cimatti, A., Nadel, A., Sebastiani, R., Shalev, J.: Applying SMT in symbolic execution of microcode. In: FMCAD (2010)
13. Freeman, T., Pfenning, F.: Refinement types for ML. In: Proc. ACM PLDI (1991)
14. Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S.: Automated termination proofs with AProVE. In: van Oostrom, V. (ed.) RTA 2004. LNCS, vol. 3091, pp. 210–220. Springer, Heidelberg (2004)
15. Gries, D.: The Science of Programming. Springer, Heidelberg (1981)
16. Haftmann, F., Nipkow, T.: A code generator framework for Isabelle/HOL. In: Theorem Proving in Higher Order Logics: Emerging Trends Proceedings (2007)
17. Jhala, R., Majumdar, R., Rybalchenko, A.: HMC: Verifying functional programs using abstract interpreters. In: Computer Aided Verification, CAV (2011)
18. Kaufmann, M., Manolios, P., Moore, J.S. (eds.): Computer-Aided Reasoning: ACL2 Case Studies. Kluwer Academic Publishers, Dordrecht (2000)
19. Kobayashi, N., Tabuchi, N., Unno, H.: Higher-order multi-parameter tree transducers and recursion schemes for program verification. In: POPL (2010)
20. Manolios, P., Turon, A.: All-termination(T). In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 398–412. Springer, Heidelberg (2009)
21. Moore, J.S.: Theorem proving for verification - the early days. In: Keynote Talk at FLoC, Edinburgh (July 2010)
22. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
23. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
24. Odersky, M.: Contracts for scala. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., Tillmann, N. (eds.) RV 2010. LNCS, vol. 6418, pp. 51–57. Springer, Heidelberg (2010)

25. Odersky, M., Spoon, L., Venners, B.: Programming in Scala: a comprehensive step-by-step guide. Artima Press (2008)
26. Rondon, P.M., Kawaguchi, M., Jhala, R.: Liquid types. In: PLDI (2008)
27. Sinha, N.: Modular bug detection with inertial refinement. In: FMCAD (2010)
28. Suter, P., Dotta, M., Kuncak, V.: Decision procedures for algebraic data types with abstractions. In: POPL (2010)
29. Taghdiri, M.: Inferring specifications to detect errors in code. In: ASE 2004 (2004)
30. VSComp: The Verified Software Competition (2010), http://www.macs.hw.ac.uk/vstte10/Competition.html
31. Walther, C., Schweitzer, S.: About veriFun. In: Baader, F. (ed.) CADE 2003. LNCS (LNAI), vol. 2741, pp. 322–327. Springer, Heidelberg (2003)
32. Zee, K., Kuncak, V., Taylor, M., Rinard, M.: Runtime checking for program verification. In: Sokolsky, O., Taşıran, S. (eds.) RV 2007. LNCS, vol. 4839, pp. 202–213. Springer, Heidelberg (2007)

Probabilistically Accurate Program Transformations

Sasa Misailovic, Daniel M. Roy, and Martin C. Rinard

MIT CSAIL
{misailo,droy,rinard}@csail.mit.edu

Abstract. The standard approach to program transformation involves the use of discrete logical reasoning to prove that the transformation does not change the observable semantics of the program. We propose a new approach that, in contrast, uses probabilistic reasoning to justify the application of transformations that may change, within probabilistic accuracy bounds, the result that the program produces.

Our new approach produces probabilistic guarantees of the form $\mathbb{P}(|D| \geq B) \leq \epsilon$, $\epsilon \in (0, 1)$, where D is the difference between the results that the transformed and original programs produce, B is an acceptability bound on the absolute value of D, and ϵ is the maximum acceptable probability of observing large $|D|$. We show how to use our approach to justify the application of loop perforation (which transforms loops to execute fewer iterations) to a set of computational patterns.

1 Introduction

The standard approach to program transformation involves the use of discrete logical reasoning to prove that the applied transformation does not change the observable semantics of the program, This paper, in contrast, introduces a novel approach that uses probabilistic reasoning to justify transformations that may change the result that the program produces.

Our approach provides probabilistic guarantees that the absolute value of the difference between the results that the transformed and original programs produce will rarely be large. A user or developer can specify bounds on the acceptable difference. The analysis can then determine the conditions under which the transformed computation satisfies the probabilistic guarantees for those bounds.

1.1 Loop Perforation

In this paper, we focus on *loop perforation*, which transforms loops to execute only a subset of their original iterations. Empirical results demonstrate the utility and effectiveness of loop perforation in reducing the amount of time (and/or other resources such as energy) that the application requires to produce a result while preserving acceptable accuracy [33,18,25,38]. While investigating the reasons behind these empirical results, we identified specific computations in our benchmark applications that interacted well with loop perforation.

E. Yahav (Ed.): SAS 2011, LNCS 6887, pp. 316–333, 2011.

Inspired by these computations, in this paper we present four generalized computational patterns that interact well with loop perforation. We have previously proposed the the use of Monte-Carlo simulation to explore how loop perforation changes the result that specific computational patterns produce [35]. In this paper we propose an analytical approach that produces algebraic expressions that characterize the effect of loop perforation on our identified set of computational patterns.

1.2 Computational Patterns

We present four computational patterns that interact well with loop perforation — the *sum* pattern, which calculates the sum of elements, the *mean* pattern, which calculates the mean of elements, the *argmin-sum* pattern, which calculates the index of the minimum sum of elements, and the *ratio* pattern, which calculates the ratio of two sums. These patterns identify general classes of computations to which we can apply the analyses that we present in this paper (rather than specific syntactic structures). The following code, for example, uses a `for` loop to implement a mean pattern. Note that the pattern abstracts away the details of the computation performed in each loop iteration, represented by a function call `f(i)`. We call each such abstracted value an *input* to the pattern.

```
sum = 0.0;
for (int i = 1; i <= n; i++) sum += f(i);
mean = sum / n;
```

In general, there may be many potential ways to realize each pattern in an actual program. In some cases the pattern may be inherent in the programming language constructs used to express the computation. For example, the sum pattern may be realized by an explicit reduction operation in a functional language. In other cases (as in the `for` loop example above) the pattern may be realized by combinations of constructs. In such cases existing program analyses (for example, reduction recognition [16,20]) can identify specific instances of these patterns.

1.3 Modeling and Analysis

We quantify the effect of loop perforation by defining the *perforation noise* – the difference between the result that the perforated computation produces and the result that the original computation produces. We denote this (signed) noise as D. The probabilistic guarantees have the form: $\mathbb{P}(|D| \geq B) \leq \epsilon$, $\epsilon \in (0, 1)$, where $\mathbb{P}(|D| \geq B)$ is the probability that the absolute value $|D|$ is greater than or equal to some bound B. The probability ϵ is the maximum acceptable probability of observing large $|D|$.

We use random variables to model our *uncertainty* about the input values and the results that subcomputations produce. We express the perforation noise as a function of these random variables. Random variables in our analyses can represent 1) inherent randomness in the inputs and/or 2) our incomplete knowledge about the exact underlying processes that produce these inputs. These two forms of uncertainty are called aleatory (objective) and epistemic (subjective) uncertainty. Our probabilistic model therefore allows us to analyze both probabilistic and deterministic computations (although in this paper we focus on deterministic computations).

We use properties of random variables, in combination with the applied transformations, to derive algebraic expressions that characterize the perforation noise. Specifically, for each pattern our analysis produces algebraic expressions that characterize the expected value and variance of the perforation noise as well as the probability of observing large absolute perforation noise. Our analysis also identifies the conditions under which these expressions are accurate. These conditions may include the number of iterations of perforated loops or distribution or independence assumptions involving random variables. Multiple analyses, each with different conditions, may be applicable to a single pattern. The expressions and the conditions that the analysis produces precisely characterize the effect of loop perforation on the result that the computation produces, providing the information that automated procedures, users, or developers need to determine whether to apply a candidate transformation.

1.4 Contributions

To the best of our knowledge, this paper is the first to propose the concept of using static program analysis to derive probabilistic accuracy bounds to justify transformations that may change the result that the program produces. It is also the first to present specific static analyses which produce such probabilistic bounds. This paper makes the following specific contributions:

- **New Paradigm:** It presents a new paradigm for justifying program transformations. Instead of using discrete logical reasoning to justify transformations that preserve the exact semantics of the program, this paradigm uses probabilistic reasoning to justify transformations that may, within guaranteed probabilistic bounds, change the result that the program produces.
- **Probabilistic Modeling of Uncertainty:** Every static analysis must somehow characterize its uncertainty about the behavior of the analyzed computation. The standard approach is to use conservative, worst-case reasoning to show that the transformation preserves the semantics in all cases.

  Our novel approach, in contrast, models the values which the computation manipulates as random variables. We use properties of these random variables to reason about the effect of the transformation on the values that the computation produces. In particular, our analysis produces derived random variables that model the difference between the result that the transformed computation produces and the result that the original computation produces.
- **Probabilistic Accuracy Guarantees:** The analysis extracts properties of the derived random variables such as their mean, variance, and probabilistic bounds on their magnitude. It uses these properties to extract probabilistic

accuracy guarantees that characterize the probability of observing unacceptably large changes in the result that the transformed program produces. It uses these guarantees to determine whether or not it is acceptable to apply the transformation.

- **Pattern-based Analyses:** It presents a set of computational patterns (*sum*, *mean*, *argmin-sum*, and *ratio*) with transformations that can be analyzed using probabilistic analyses. For each pattern it presents a probabilistic analysis that characterizes the effect of loop perforation on the result that the pattern produces. Each analysis produces expressions for the expected value and variance of the absolute perforation noise and bounds that characterize the probability of observing large perforation noise.

2 Example

Swaptions is a financial analysis application from the PARSEC benchmark suite [4]; it uses Monte Carlo simulation to calculate the price of a portfolio of swaption financial instruments. Figure 1(a) presents an abstract version of a perforatable computation from this application. The loop performs a sequence of simulations to compute the mean simulated value of the swaption into the variable `dMeanPrice`. The analysis recognizes this computation as an instance of the *mean* pattern.

Figure 1(b) presents the transformed version of the computation after loop perforation [18], applied in this case by changing the induction variable increment from `1` to `2`. The perforated loop therefore executes half of the iterations of the original loop. The transformed computation also extrapolates the result to eliminate any systematic bias introduced by executing fewer loop iterations.

Modeling Values With Random Variables. The analysis models the value of `simres` at each iteration as a random variable X_i, $i \in \{1, \dots n\}$. The variable `dPrice` contains the sum of the X_i. In the original computation, the final value of `dPrice` is $S_O = \sum_{i=1}^{n} X_i$. In the perforated computation the final value is $S_P = \sum_{i=1}^{n/2} X_{2i-1}$ (for simplicity, we present the analysis for the case when `n` is even). After extrapolating S_P, the perforation noise $D = \frac{1}{n}\big(2S_P - S_O\big)$.

Scenarios. We have developed analyses for a variety of scenarios, with each scenario characterized by properties such as the distributions of the X_i and the

```
float dPrice = 0.0;
for (i = 1; i <= lTrials; i += 1) {
   float simres = runSimulation(this, i)
   dPrice += simres;
}
double dMeanPrice = dPrice / lTrials;
printf("%g\n", dMeanPrice);
```

(a) Original Computation

```
float dPrice = 0.0;
for (i = 1; i <= lTrials; i += 2) {
   float simres = runSimulation(this, i)
   dPrice += simres;
}
double dMeanPrice = (dPrice * 2) / lTrials;
printf("%g\n", dMeanPrice);
```

(b) Transformed Computation

Fig. 1. Swaptions Computation: Original and Transformed Computations

number of loop iterations n. For each scenario the analysis produces an ϵ and B such that $\mathbb{P}(|D| \geq B) \leq \epsilon$, $\epsilon \in (0, 1)$. Specific scenarios for which analyses exist include (see Section 4) (a) the X_i have finite mean and covariance, (b) the X_i are independent and identically distributed (i.i.d.), (b.1) in addition, the number of iterations n of the original loop is large, or (b.2) n may be small but the X_i are normally distributed, or (c) the X_i are correlated and form a random walk. Note that some scenarios are more specific than others; in general, analyses for more specific scenarios tend to deliver tighter probabilistic bounds.

Scenario (b.1). We next discuss how the analysis proceeds for Scenario (b.1) (the X_i are i.i.d. and n is large). Using results from Section 4, the expected value of the perforation noise $\mathbb{E}(D) = 0$, and the variance is $\mathrm{var}(D) = \sigma^2\left(\frac{1}{m} - \frac{1}{n}\right) = \frac{\sigma^2}{n}$, where σ^2 is the variance of the input variables X_i.

Because S_P and $S_O - S_P$ are sums of a large number of i.i.d. random variables, the distribution of their means approaches a normal distribution (by a Central Limit Theorem argument). Furthermore, since D is a linear combination of these two independent means, D is also normally distributed and we can use the Gaussian confidence interval to obtain the following bound: with probability at least $1 - \epsilon$, $|D| < z_{1-\frac{\epsilon}{2}}\sqrt{\mathrm{var}(D)} = z_{1-\frac{\epsilon}{2}}\frac{\sigma}{\sqrt{n}}$, where z_α is the quantile function of the standard normal distribution. For example, $\mathbb{P}(|D| \geq B) \leq 0.05$ for $B \approx \frac{1.96\sigma}{\sqrt{n}}$ ($z_{0.975} \approx 1.96$).

Comparison with Worst-Case Analysis. We next compare the probabilistic and worst-case analyses in an identical scenario. We assume that the X_i are i.i.d. random variables drawn from the uniform distribution on $[a, b]$. In this case the 95% probabilistic bound is $B_P = 1.96\frac{b-a}{\sqrt{12n}} \approx 0.6\frac{b-a}{\sqrt{n}}$ ($\sigma^2 = \frac{(b-a)^2}{12}$ for the uniform distribution). The worst-case analysis, on the other hand, extracts the bound $B_W = \frac{b-a}{2}$. Note that the worst case bound B_W is asymptotically $\sqrt{n}$ times larger than the probabilistic bound B_P.

Choosing an Appropriate Scenario. In general, the validity of the probabilistic bounds may depend on how closely the actual use case matches the selected analysis scenario. In some cases it may be appropriate to choose a scenario by recording values from instrumented representative executions of the computation, then using the values to (potentially conservatively) select aspects of the scenario such as specific probability distributions or the expected number of loop iterations [24]. In other cases it may be appropriate to have a user or developer simply provide this information directly to the analysis [24].

Identifying Appropriate ϵ and B. In general, acceptable values for ϵ and B will depend on the application and the context in which it is used. We therefore anticipate that the user (or potentially the developer) will identify the maximum acceptable ϵ and B. Transformations with ϵ and B less than these maxima are acceptable. Transformations with ϵ or B greater than these maxima are unacceptable.

3 Preliminaries

We next describe the notation that we use throughout the probabilistic analyses. The original loop executes n iterations; the perforated loop executes m, $m < n$ iterations. The *perforation rate* $r = 1 - \lfloor \frac{m}{n} \rfloor$. A loop perforation *strategy* can be represented by a $n \times 1$ *perforation vector* P, each element of which corresponds to a single loop iteration. The number of non-zero elements of P is equal to m. The all-ones perforation vector $A = (1, \ldots, 1)'$ represents the original (non-perforated) computation. Some transformations may use the perforation vector. The perforation transformation for the sum pattern, for example, uses the values of the non-zero elements of the vector P to extrapolate the final result.

Interleaving perforation executes every k-th iteration, where $k = \lfloor \frac{n}{m} \rfloor$. The corresponding perforation vector has elements $P_{ki+1} = 1$, where $i \in \{0, ...m-1\}$, and $P_{ki+j} = 0$ for $j < k, j \neq 1$. *Truncation perforation* executes m iterations at the beginning of the loop; the perforation vector has elements $P_i = 1$ for $1 \leq i \leq m$, and $P_i = 0$ otherwise. *Randomized perforation* selects a random subset of m elements. All of these perforation strategies can be implemented efficiently without explicitly creating the vector P.

4 Patterns and Analyses

For each pattern we present an example of the original and the transformed code. For simplicity we apply an interleaving perforation and assume a number of iterations `n` to be a multiple of the new loop increment `k`.

In each pattern analysis section we first present the assumptions we make on the distribution of the inputs. These assumptions characterize our uncertainty about the values of these inputs. With these assumptions in place, we derive expressions for 1) the mean perforation noise, 2) the variance of the perforation noise, and 3) bounds on the probability of observing large absolute perforation noise. In some cases we perform additional analyses based on additional assumptions. When applicable, we present bounds based on Chebyshev's inequality, Hoeffding's inequality, and Gaussian confidence intervals. We present a more detailed derivation of these expressions in our accompanying technical report [24].

4.1 Sum Pattern

We present an example of the original and perforated code for the extrapolated sum pattern in Figure 2. We first present a generalized analysis for the sum

Original code	Transformed Code

`double sum = 0.0;`
`for (int i = 1; i <= n; i++) {`
`  sum += f(i);`
`}`
	`double sum = 0.0;`
	`for (int i = 1; i <= n; i+=k) {`
	`  sum += f(i);`
	`}`
	`sum *= k;`

Fig. 2. Sum Pattern; Original and Transformed Code

of correlated random variables. We then present specializations of the analysis under additional assumptions. Special cases that we analyze include independent and identically distributed (i.i.d.) inputs and inputs generated by a random walk.

Assumptions. We first assume only that the terms of the sum have a common finite mean μ and finite covariance.

Analysis. For $i = 1, \ldots, n$, let $X_i = \texttt{f}(i)$ be the i-th term of the summation. We model our uncertainty about the values X_i by treating $X = (X_1, \ldots, X_n)'$ as a vector of n random variables with mean μ and covariance matrix Σ with elements $(\Sigma)_{ij} = \operatorname{cov}(X_i, X_j)$. Let A be the all-ones vector defined in Section 3, then $A'X = \sum_{i=1}^{n} X_i$. Let P be a perforation vector with m non-zero elements. Then $P'X = \sum_{i=1}^{n} P_i X_i$ is the result of the perforated computation. The signed perforation noise is $D \equiv P'X - A'X = (P - A)' X$ with

$$\mathbb{E}(D) = \mu \sum_{i=1}^{n} (P_i - 1), \tag{1}$$

$$\operatorname{var}(D) = \sum_{i,j} (P_i - 1)(P_j - 1)\, \Sigma_{i,j}. \tag{2}$$

To avoid systematic bias, we can choose P so that $\mathbb{E}(D) = 0$. In particular, it follows from Equation 1 that an estimate is *unbiased* if and only if $\sum_{i=1}^{n} P_i = n$. One extrapolation strategy equally extrapolates every non-zero element, choosing $P_i = \frac{n}{m}$ for non-zero elements P_i.

If P satisfies $\mathbb{E}(D) = 0$, we can use Chebyshev's inequality and $\operatorname{var}(D)$ to bound the absolute perforation noise, such that, with probability at least $1 - \epsilon$

$$|D| < \sqrt{\frac{\operatorname{var}(D)}{\epsilon}} \tag{3}$$

This bound will be conservative in practice; additional knowledge (e.g., independence or distribution of X_i) can be used to derive tighter bounds. We next study a number of special cases in which additional assumptions enable us to better characterize the effect of perforation.

Independent Variables

Assumptions. We assume that the elements $X_i = \texttt{f}(i)$ of the summation are i.i.d. random variables with finite mean μ and variance σ^2. To derive a tighter bound on the mean and the variance of the absolute perforation noise, we consider additional assumptions – specifically, that the X_i are normally distributed, or that the X_i are bounded.

Analysis. From (1), we know that $\mathbb{E}(D) = 0$ for any perforation P such that $\sum_i P = n$. From (2), and since the covariance matrix Σ of i.i.d. variables has non-zero values only along its leading diagonal, it follows that $\operatorname{var}(D) = \sigma^2 \sum_i (1 - P_i)^2$. It is straightforward to show that this value is minimized by any perforation vector P with $n - m$ zeros and the remaining elements taking the value $\frac{n}{m}$. In this case, the variance takes the value

$$\mathrm{var}(D) = \frac{\sigma^2\, n\,(n-m)}{m}. \tag{4}$$

We can immediately bound the probability of observing large absolute perforation noise using Chebyshev's inequality (Equation 3).

We can get potentially tighter bounds if we make additional assumptions. If we assume that each term X_i is normally distributed, then D will also be normally distributed. Consequently, $\mathbb{E}(D) = 0$ and $\mathrm{var}(D)$ remains the same as in Equation 4.

The normality of D allows us to obtain a tighter bound on the perforation noise. In particular, with probability $1-\epsilon$

$$|D| \leq z_{1-\frac{\epsilon}{2}} \sqrt{\mathrm{var}(D)} \tag{5}$$

where z_α is the quantile function of the standard normal distribution. As a comparison, for $\epsilon = 0.01$ the bound (5) is 6.6 times smaller than the Chebyshev-style bound (3). For normally distributed D we can also bound the absolute perforation noise. In particular, $|D|$ has a half-normal distribution with mean $\mathbb{E}\Big(|D|\Big) = \sigma\sqrt{\frac{2n(n-m)}{\pi m}}$, and variance $\mathrm{var}(|D|) = \Big(1 - \frac{2}{\pi}\Big)\,\mathrm{var}(D)$.

If, instead, we assume that each X_i is bounded, falling in the range $[a, b]$, we can apply Hoeffding's inequality to bound the absolute perforation noise $|D|$. Let $X'_i = (P_i - 1)X_i$, and note that the variables X'_i are also mutually independent. The range of X'_i is $[a_i, b_i] = \big[(P_i-1)a, (P_i-1)b\big]$. Then the sum $\sum_{i=1}^{n}(b_i - a_i)^2 = (b-a)^2\frac{n\,(n-m)}{m}$, and thus, with probability at least $1-\epsilon$

$$|D| < \sqrt{\frac{1}{2}\ln\frac{2}{\epsilon}\cdot\sum_{i=1}^{n}\Big(b_i - a_i\Big)^2} = (b-a)\sqrt{\frac{n\,(n-m)}{2m}\ln\frac{2}{\epsilon}}. \tag{6}$$

Nested Loops. We next extend the sum analysis (for i.i.d. inputs) to nested loops. The outer loop executes n_1 iterations (m_1 iterations after perforation); the inner loop executes n_2 iterations (m_2 iterations after perforation). When both loops are perforated, $\mathbb{E}(D) = 0$. We use Equation 4 to compute $\mathrm{var}(D)$ by assigning $n = n_1 n_2$ and $m = m_1 m_2$. The result generalizes to more deeply nested loops.

Random Walk

Assumptions. We assume that the sequence X of random variables is a random walk with independent increments. Specifically, we assume that the sequence is a Markov process, and that the differences between the values at adjacent time steps $\delta_i = X_{i+1} - X_i$ are a sequence of i.i.d. random variables with mean 0 and variance σ^2. Let $X_0 = \mu$ be a constant.

Analysis. From the assumption $\mathbb{E}(\delta_i) = 0$, it follows by induction that the expected value of every element is $\mathbb{E}(X_i) = \mu$. As a consequence, for any perforation vector that satisfies $\sum_{i=1}^{n} P_i = n$, we have that $\mathbb{E}(D) = 0$.

For $i < j$, the covariance between X_i and X_j satisfies $cov(X_i, X_j) = i\sigma^2$. Therefore, the covariance matrix Σ has entries $(\Sigma)_{ij} = \sigma^2 \min\{i, j\}$, and the variance of the perforation noise satisfies

$$\mathrm{var}(D) = \sigma^2 \sum_{i,j} (1 - P_i)(1 - P_j) \min\{i, j\}. \tag{7}$$

We may choose a perforation strategy P by minimizing this variance (and thus minimizing Chebyshev's bound on the absolute perforation noise). For example, when $P_i = 2$ for odd i and 0 otherwise, we have that $\mathrm{var}(D) = \frac{n}{2}\sigma^2$. Once again, we can use Chebyshev's inequality or Gaussian confidence intervals to derive a probabilistic accuracy bound.

4.2 Mean Pattern

We present an example of the original and perforated code for the mean pattern in Figure 3.

Original code	Transformed Code

`double sum = 0.0;`
`for (int i = 1; i <= n; i++) {`
`  sum += f(i);`
`}`
`double mean = sum / n;`
	`double sum = 0.0;`
	`for (int i = 1; i <= n; i+=k) {`
	`  sum += f(i);`
	`}`
	`double mean = sum * k / n;`

Fig. 3. Mean Pattern; Original and Transformed Code

We can extend the analysis for the sum pattern (Section 4.1) because the result of the mean computation is equal to the result of the sum computation divided by n. We denote the perforation noise of the sum as D_{Sum}, the output produced by the original computation as $\frac{1}{n}A'X$, and the output produced by the perforated computation as $\frac{1}{n}P'X$. The perforation noise of the mean D in the general case with correlated variables is $D \equiv \frac{1}{n}\big(P'X - A'X\big) = \frac{1}{n}D_{Sum}$. By the linearity of expectation, the perforation noise has expectation $\mathbb{E}(D) = \frac{1}{n}\mathbb{E}(D_{Sum})$ and variance

$$\mathrm{var}(D) = \frac{1}{n^2}\,\mathrm{var}(D_{Sum}). \tag{8}$$

The derivation of the bounds for the more specific cases (i.i.d., normal, random walk inputs) is analogous to the derivation discussed in Section 4.1. In particular if we assume i.i.d. inputs, the variance $\mathrm{var}(D) = \sigma^2\big(\frac{1}{m} - \frac{1}{n}\big)$. Based on Chebyshev's and Hoeffding's inequalities, we can derive algebraic expressions that characterize the probabilistic accuracy bounds for this case. A similar result can be shown for the random walk case.

We can also obtain a potentially tighter Gaussian interval style bound if we assume a large number of i.i.d. inputs with finite mean and variance. In this case the sums $P'X$ and $(A - \frac{m}{n}P)'X$ will be independent and their means will be approximately normally distributed (by a Central Limit Theorem argument).[1] Consequently, the perforation noise D, which is a linear combination of these two means, will also be approximately normally distributed. We can then use Equation 5 to calculate a bound on the perforation noise.

4.3 Argmin-Sum Pattern

We present an example of the original and transformed code for the argmin-sum pattern in Figure 4.[2]

Original code

```
double best = MAX_DOUBLE;
int best_index = -1;
for (int i = 1; i <= L; i++) {
  s[i] = 0;
  for (int j = 1; j <= n; j++)
    s[i] += f(i,j);

  if (s[i] < best) {
     best = s[i];
     best_index = i;
  }
}
return best_index;
```

Transformed Code

```
double best = MAX_DOUBLE;
int best_index = -1;
for (int i = 1; i <= L; i++) {
  s[i] = 0;
  for (int j = 1; j <= n; j+=k)
    s[i] += f(i,j);

  if (s[i] < best) {
     best = s[i];
     best_index = i;
  }
}
return best_index;
```

Fig. 4. Argmin-Sum Pattern; Original and Transformed Code

Assumptions. For each $i \in \{1, \ldots, L\}$, we assume that $X_{i,j} = \mathtt{f(i,j)}$ are independent and drawn from some distribution F. The elements of the perforation vector P take only the values from the set $\{0, 1\}$.

Analysis The output of the argmin-sum pattern is an index which is used later in the program. To calculate the perforation noise, we model the *weight* of an index i as the entire sum $X_i = \sum_{j=1}^{n} X_{i,j}$. Therefore, the original computation produces the value $S_O = \min_i A'X_i = \min_i \sum_{j=1}^{n} X_{i,j}$, while the perforated computation produces the value $S_P = \sum_{j=1}^{n} X_{\gamma,j}$, where $\gamma \equiv \arg\min_i \sum_{j=1}^{m} X_{i,j}$ and m is the reduced number of steps in the perforated sum. Note that the independence of the variables $X_{i,j}$ implies that we can, without loss of generality, choose any perforation strategy with perforation rate r.

[1] Note that the tails of the distribution of the mean (which we use to bound the perforation noise) converge to the normal distribution slower than the means. The Berry-Esseen inequality can be used to determine how closely the normal distribution approximates the actual distribution of the sum. In particular, if n is the number of the terms, the maximum distance between the standardized sum distribution and the normal distribution is less than $\delta \approx 0.48 \frac{\rho}{\sigma^3 \sqrt{n}}$, where $\rho = \mathbb{E}(|X_i|^3)$ and $\sigma^2 = \mathrm{var}(X_i)$.

[2] We can apply the same analysis to the *min-sum* pattern, which returns the (extrapolated) value `best` instead of the index `best_index`. It is also possible to modify this analysis to support the *max-sum* and *argmax-sum* patterns.

We are interested in studying the perforation noise $D \equiv S_P - S_O$. Note that the perforation noise D is non-negative because S_O is a minimum sum, and so $D = |D|$ is also the absolute perforation noise.

Let $Y_i \equiv \sum_{j=1}^{m} X_{i,j}$ and $Z_i \equiv \sum_{j=m+1}^{n} X_{i,j}$. Then, $S_O = \min_i (Y_i + Z_i) = Y_\omega + Z_\omega$ and $S_P = Y_\gamma + Z_\gamma$ where $\gamma = \arg\min_i Y_i$ and $\omega = \arg\min_i (Y_i + Z_i)$ is the index of the minimum sum. Then the perforation noise satisfies

$$D \leq Z_\gamma - \min_i Z_i. \tag{9}$$

Let $\bar{D} \equiv Z_\gamma - \min_i Z_i$ denote this upper bound. We can obtain conservative estimates of the perforation noise D by studying $\bar{D}$. Note that for this pattern, $\bar{D}$ is always non-negative (because $Z_\gamma \geq \min_i Z_i$).

Let Z be a sum of $n - m$ independent F-distributed random variables. Then (1) Z_i has the same distribution as Z, (2) γ is independent of Z_γ, and (3) Z_γ has the same distribution as Z. Therefore,

$$\mathbb{E}(\bar{D}) = \mathbb{E}(Z) - \mathbb{E}(\min_i Z_i), \tag{10}$$

or, put simply, the expectation of our bound $\bar{D}$ is the difference between the mean of Z and its first order statistic (given a size L sample).

To proceed with the analysis, we make the additional assumption that Z is uniformly distributed on the interval $a \pm \frac{w}{2}$ of width $w > 0$ and center a.[3] Let Z_i be i.i.d. copies of Z.

Define $M_L = \min_{i \leq L} Z_i$. Then $\frac{1}{w}(M_L - a + \frac{w}{2})$ has a Beta$(1, L)$ distribution, and so $\mathbb{E}(M_L) = a + \frac{w}{L+1} - \frac{w}{2}$ and variance $\mathrm{var}(M_L) = \frac{Lw^2}{(L+1)^2(L+2)}$. From (10), we have $\mathbb{E}(\bar{D}) = \frac{w}{2} - \frac{w}{L+1}$. Furthermore, as γ is independent of every Z_i, it follows that Z_γ is independent of M_L. Therefore,

$$\mathrm{var}(\bar{D}) = \frac{1}{12}w^2 + \frac{Lw^2}{(L+1)^2(L+2)}. \tag{11}$$

The mean and variance of $\bar{D}$ can be used to derive one-sided Chebyshev style bounds on $\bar{D}$ and, since $|D| = D < \bar{D}$, bounds on the absolute perforation noise $|D|$. In particular, using Chebyshev's one-sided inequality, it follows that with probability at least $1 - \epsilon$

$$|D| < \sqrt{\mathrm{var}(\bar{D})\Big(\frac{1}{\epsilon} - 1\Big)} + \mathbb{E}\bar{D} \tag{12}$$

4.4 Ratio Pattern

We present an example of the original and transformed code for the ratio pattern in Figure 5.

[3] We anticipate that Z will in practice rarely be uniform, however this assumption simplifies the analysis and is in some sense conservative if we choose the center and width to cover all but a tiny fraction of the mass of the true distribution of Z. Note that when, instead, Z is Gaussian, the variance of the perforation noise does not have a closed form [2]. However, if we assume Z to be uniform, we might take our approximation to cover some number of standard deviations.

Original code	Transformed Code

double numer = 0.0;
double denom = 0.0;
for (int i = 1; i <= n; i++) {
  numer += x(i);
  denom += y(i);
}
return numer/denom;
	double numer = 0.0;
	double denom = 0.0;
	for (int i = 1; i <= n; i+=k) {
	  numer += x(i);
	  denom += y(i);
	}
	return numer/denom;

Fig. 5. Ratio Pattern; Original and Transformed Code

Assumptions. Let $X_i = \mathtt{x}(i)$ and $Y_i = \mathtt{y}(i)$ denote random variables representing the values of the inner computations. We assume that the sequence of pairs (X_i, Y_i) are i.i.d. copies of a pair of random variables (X, Y), where $Y > 0$ almost surely. Define $Z = X/Y$ and $Z_i = X_i/Y_i$. For some constants μ and σ_Z^2, we assume that the conditional expectation of Z given Y is μ, i.e., $\mathbb{E}(Z|Y) = \mu$, and that the conditional variance satisfies $\mathrm{var}(Z|Y) = \frac{\sigma_Z^2}{Y}$.

Analysis. The elements of the perforation vector P only take values from the set $\{0, 1\}$. Note that the independence of the pairs (X_i, Y_i) from different iterations implies that the perforation strategy does not influence the final result. To simplify the derivation, but without loss of generality, we use the perforation vector P in which the first m elements are 1 and the remaining elements 0.

Define $Y_1^n = A'Y = \sum_{i=1}^n Y_i$ and $Y_1^m = P'Y = \sum_{i=1}^m Y_i$ and define X_1^n and X_1^m analogously. Then the value of the original computation is $S_O = \frac{X_1^n}{Y_1^n} = \sum_{i=1}^n \frac{Y_i}{Y_1^n} Z_i$, while the value of the perforated computation is given by $S_P = \sum_{i=1}^m \frac{Y_i}{Y_1^m} Z_i$, where m is the reduced number of steps in the perforated sum. Note that in the previous equations, we used the identity $X_i = Y_i Z_i$.

We begin by studying the (signed) perforation noise $D \equiv S_P - S_O$. The conditional expectation of D given $Y_{1:n} = \{Y_1, \ldots, Y_n\}$ satisfies $\mathbb{E}(D|Y_{1:n}) = \sum_{i=1}^n \frac{Y_i}{Y_1^n} \mu - \sum_{i=1}^m \frac{Y_i}{Y_1^m} \mu = 0$. The conditional variance satisfies $\mathrm{var}(D|Y_{1:n}) = \sigma_Z^2 \left(\frac{1}{Y_1^m} - \frac{1}{Y_1^n} \right)$ By the law of iterated expectations $\mathbb{E}(D) = \mathbb{E}(\mathbb{E}(D|Y_{1:n})) = 0$.

To proceed with an analysis of the variance of the perforation noise D, we make a distributional assumption on Y. In particular, we assume that Y is gamma distributed with shape $\alpha > 1$ and scale $\theta > 0$. Therefore, the sum Y_1^m also has a gamma distribution with parameters $\alpha' = m\alpha$, $\theta' = \theta$, $\frac{1}{Y_1^m}$ has an inverse gamma distribution with mean $(\theta(m\alpha - 1))^{-1}$, and so

$$\mathrm{var}(D) = \frac{\sigma_Z^2}{\theta} \left(\frac{1}{m\alpha - 1} - \frac{1}{n\alpha - 1} \right). \tag{13}$$

Again, using Chebyshev's inequality, we can bound the probability of large absolute perforation noise $|D|$.

5 Discussion

Usage Scenarios. We anticipate several usage scenarios for the analyses we present in Section 4. First, the analyses can provide the formal foundation required to justify the automatic application of loop perforation. In this scenario, the analysis works with a probabilistic accuracy specification (which provides the desired accuracy bounds and probability with which the transformed computation should satisfy the bounds) and a specification of the probability distributions for the random variables used to model pattern inputs. These probability distributions can be provided either by a developer, by a user, or by fitting distributions to values observed during profiling executions. In [24] we present an initial empirical evaluation of our probabilistic analyses on perforatable computations from the PARSEC benchmark suite.

Second, the analyses can also help users and developers better understand the effect of loop perforation. They may then use this information to select an optimal operating point for their application given their combined performance and accuracy constraints and requirements.

Third, the analyses can also help a control system dynamically select optimal application operating points as underlying characteristics (such as load, clock rate, number of processors executing the computation, or any other characteristic that many affect the delivered computational capacity) of the underlying computational platform change [18,33].

In all of these scenarios the probabilistic analyses in this paper can be used to better understand the shape of the trade-off space and more productively drive the selection of perforation policies, with appropriate maximum acceptable ϵ and B determining the range of available policies.

Scope. In this paper we provide probabilistic guarantees for the accuracy of perforated computations. We expect that the basic framework of the probabilistic guarantees (algebraic expressions for expected values, variances, and probabilistic bounds) will remain largely the same for other transformations (the derivation of the expressions will of course differ). We note that even for loop perforation, we do not attempt to provide an exhaustive list of the possible patterns and analyses. The statistical literature provides a comprehensive treatment of operations on random variables [41] and order statistics of random variables [2]. The basic compositional properties of probability distributions under such operations can provide the foundation for the analysis of computations which employ many of these operations. In addition, for the random perforation strategy, survey sampling [9] can provide useful bounds that do not make assumptions on the distribution of the inputs for a number of aggregation computations.

We note that, given known composition properties of operations on probability distributions (for example, sums of Gaussian distributions are Gaussian; multiplying a Gaussian by constant produces another Gaussian), it is possible to compose our pattern-based analyses in straightforward ways to analyze more complex computations. For example, it is straightforward to generalize the analysis of the sum pattern to analyze arbitrary linear computations over values modeled using Gaussian distributions.

We also anticipate that a number of program analyses or type systems can work in concert with our probabilistic analyses. These analyses and type systems can, for example, help identify computational patterns, increase confidence in some of the input assumptions, or reason about safety of the transformation. For example, analyses or type systems may distinguish critical parts of the computation (which, if transformed, can dramatically change the behavior of the application and as such should not be perforated), from approximate parts of the computation, which can be perforated [5,37].

6 Related Work

Loop Perforation and Task Skipping: Loop perforation [18,25,38] can be seen as a special case of task skipping [33,34]. The first publication on task skipping used linear regression to obtain empirical statistical models of the time and accuracy effects of skipping tasks and identified the use of these models in purposefully skipping tasks to reduce the amount of resources required to perform the computation while preserving acceptable accuracy [33].

The first publication on loop perforation presented a purely empirical justification of loop perforation with no formal statistical, probabilistic, or discrete logical reasoning used to justify the transformation [18]. The first statistical justification of loop perforation used Monte Carlo simulation [35]. The first probabilistic justification for loop perforation used a pattern-based static analysis and also presented the use of profiling runs on representative inputs and developer specifications to obtain the required probability distribution information [24]. The probabilistic analyses in this paper can help users or developers better understand the shape of the induced accuracy vs. performance trade-off space and select optimal operating points within this space given their combined accuracy and performance requirements and/or constraints.

Continuity, Sensitivity, and Robustness: Chaudhuri et al. present a program analysis for automatically determining whether a function is continuous [6]. The reasoning is deterministic and worst-case. An extension of this research introduces a notion of function robustness, and, under an input locality condition, presents an approximate memoization approach similar to loop perforation [7]. For a special case when the inputs form a Gaussian random walk (as described in Section 4.1) and the loop body is a robust function, the paper derives a probabilistic bound to provide a justification for applying loop perforation.

Reed and Pierce present a type system for capturing function sensitivity, which measures how much a function may magnify changes to its inputs [32]. Although the language contains probabilistic constructs, the type system uses deterministic worst-case reasoning, resulting in a worst-case sensitivity bound.

Modeling Uncertainty. Typical approaches for modeling uncertainty involve the use of intervals, random variables, or fuzzy sets to represent values, and the definition of computational patterns that operate on these uncertain values.

Interval analysis [28] represents uncertain values as intervals and defines basic arithmetic operations on such values. It is often used to analyze the worst-case rounding error in numerical computations, ideally producing small interval sizes. For loop perforation the interval sizes are typically much larger and the derived bounds therefore much less precise.

Additional knowledge about the inputs can make it possible to use probabilistic, fuzzy, or hybrid modeling of the computations [21] to provide tighter bounds. In this paper we use random variables to model uncertainty. The source of this uncertainty can be either 1) innate randomness in the inputs or computation or 2) our partial understanding of parts of the computation. Fuzzy or hybrid approaches to modeling uncertainty may also, at least in principle, provide alternate justifications for loop perforation.

Probabilistic Languages and Analyses: Researchers have previously defined languages for probabilistic modeling, in which programs work directly with probability distributions [36,22,19,31,11,30,13], and analyses to reason about probabilistic programs [29,12,26,27,23,10,39]. Researchers have also used a probabilistic foundation to quantitatively reason about certain properties of deterministic programs [14,8]. Our approach quantitatively analyzes the application of loop perforation to a set of amenable computational patterns, which may appear in deterministic or probabilistic programs. It specifies probabilistic semantics at a pattern level instead of the statement level. In comparison with general probabilistic analyses, pattern-based analyses can, typically, provide more precise accuracy bounds, since patterns provide additional information about the nature of the analyzed computations (instances of patterns). Moreover, patterns can identify additional information such as a definition of perforation noise (e.g., for the argmin-sum pattern), which may be impossible for a general probabilistic semantics to capture. Computational patterns similar to ours have also been used to provide more precise analyses in other contexts [15].

Performance vs. Accuracy Trade-Off Management: Both task skipping [33,34] and loop perforation [18,38] can augment an application with the ability to operate at multiple points in an underlying accuracy vs. performance trade-off space. Of particular interest is the ability to move between points in this space as the application is executing, enabling the application to adapt to the underlying computing environment [18,33,34]. The empirical discovery of Pareto-optimal combinations of perforated computations can enable a control system to find and exploit optimal operating points within the trade-off space [18].

Dynamic Knobs converts static application configuration parameters into dynamic control variables, which the system can use to change the point in the underlying trade-off space at which the application executes [17]. Eon [40], Green [3], and Petabricks [1] allow developers to provide multiple implementations of a specific piece of application functionality, with each implementation potentially exhibiting different performance versus accuracy trade-offs. There is no explicit reasoning to justify the acceptability of the different alternatives – all of these systems empirically evaluate the alternatives and ultimately rely on the developer to identify only acceptable implementations.

7 Conclusion

Traditional program analysis and transformation approaches use worst-case logical reasoning to justify the application of transformations that do not change the result that the program produces. We propose instead to use probabilistic reasoning to justify the application of transformations that may, within probabilistic bounds, change the result that the program produces. A goal is to provide a reasoning foundation that can enable the application of a richer class of program transformations.

Our results demonstrate how to apply this approach to justify the use of loop perforation, which transforms the program to skip loop iterations. We identify computations that interact well with loop perforation and show how to use probabilistic reasoning to bound how much loop perforation may change the result that the program produces. This reasoning can provide the foundation required to understand, predict, and therefore justify the application of loop perforation. In the future, we anticipate the use of similar probabilistic reasoning to justify the application of a broad range of new transformations that may change the result that the program produces.

Acknowledgements. We would like to thank Michael Carbin and the anonymous reviewers for their useful comments on the earlier drafts of this paper.

This research was supported in part by the National Science Foundation (Grants CCF-0811397, CCF-0905244, CCF-1036241 and IIS-0835652) and the United States Department of Energy (Grant DE-SC0005288).

References

1. Ansel, J., Chan, C., Wong, Y., Olszewski, M., Zhao, Q., Edelman, A., Amarasinghe, S.: Petabricks: A language and compiler for algorithmic choice. In: PLDI 2010 (2010)
2. Arnold, B., Balakrishnan, N., Nagaraja, H.: A first course in order statistics. Society for Industrial Mathematics, Philadelphia (2008)
3. Baek, W., Chilimbi, T.: Green: A framework for supporting energy-conscious programming using controlled approximation. In: PLDI 2010 (2010)
4. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The PARSEC benchmark suite: Characterization and architectural implications. In: PACT 2008 (2008)
5. Carbin, M., Rinard, M.: Automatically Identifying Critical Input Regions and Code in Applications. In: ISSTA 2010 (2010)
6. Chaudhuri, S., Gulwani, S., Lublinerman, R.: Continuity analysis of programs. In: POPL 2010 (2010)
7. Chaudhuri, S., Gulwani, S., Lublinerman, R., Navidpour, S.: Proving Programs Robust. In: FSE 2011 (2011)
8. Chaudhuri, S., Solar-Lezama, A.: Smooth interpretation. In: PLDI 2010 (2010)
9. Cochran, W.G.: Sampling techniques. John Wiley & Sons, Chichester (1977)
10. Di Pierro, A., Hankin, C., Wiklicky, H.: A systematic approach to probabilistic pointer analysis. In: ASPLAS 2007 (2007)
11. Di Pierro, A., Hankin, C., Wiklicky, H.: Probabilistic λ-calculus and quantitative program analysis. Journal of Logic and Computation (2005)

12. Di Pierro, A., Wiklicky, H.: Concurrent constraint programming: Towards probabilistic abstract interpretation. In: PPDP 2000 (2000)
13. Goodman, N., Mansinghka, V., Roy, D., Bonawitz, K., Tenenbaum, J.: Church: a language for generative models. In: UAI 2008 (2008)
14. Gulwani, S., Necula, G.C.: Precise interprocedural analysis using random interpretation. In: POPL 2005 (2005)
15. Gulwani, S., Zuleger, F.: The reachability-bound problem. In: PLDI 2010 (2010)
16. Hall, M., Murphy, B., Amarasinghe, S., Liao, S., Lam, M.: Interprocedural analysis for parallelization. In: Languages and Compilers for Parallel Computing (1996)
17. Hoffman, H., Sidiroglou, S., Carbin, M., Misailovic, S., Agarwal, A., Rinard, M.: Dynamic knobs for power-aware computing. In: ASPLOS 2011 (2011)
18. Hoffmann, H., Misailovic, S., Sidiroglou, S., Agarwal, A., Rinard, M.: Using Code Perforation to Improve Performance, Reduce Energy Consumption, and Respond to Failures. Technical Report MIT-CSAIL-TR-2009-042 (2009)
19. Hurd, J.: A formal approach to probabilistic termination. In: Carreño, V.A., Muñoz, C.A., Tahar, S. (eds.) TPHOLs 2002. LNCS, vol. 2410, p. 230. Springer, Heidelberg (2002)
20. Kennedy, K., Allen, J.R.: Optimizing compilers for modern architectures: a dependence-based approach. Morgan Kaufmann, San Francisco (2002)
21. Klir, G.: Uncertainty and information. John Wiley & Sons, Chichester (2006)
22. Kozen, D.: Semantics of probabilistic programs. Journal of Computer and System Sciences (1981)
23. Kwiatkowska, M., Norman, G., Parker, D.: Prism: Probabilistic symbolic model checker. In: Computer Performance Evaluation: Modelling Techniques and Tools (2002)
24. Misailovic, S., Roy, D., Rinard, M.: Probabilistic and Statistical Analysis of Perforated Patterns. Technical Report MIT-CSAIL-TR-2011-003, MIT (2011)
25. Misailovic, S., Sidiroglou, S., Hoffmann, H., Rinard, M.: Quality of service profiling. In: ICSE 2010 (2010)
26. Monniaux, D.: Abstract interpretation of probabilistic semantics. In: SAS 2000. LNCS, vol. 1824, pp. 322–340. Springer, Heidelberg (2000)
27. Monniaux, D.: An abstract monte-carlo method for the analysis of probabilistic programs. In: POPL 2001 (2001)
28. Moore, R.E.: Interval analysis. Prentice-Hall, Englewood Cliffs (1966)
29. Morgan, C., McIver, A.: pGCL: formal reasoning for random algorithms. South African Computer Journal 22 (1999)
30. Park, S., Pfenning, F., Thrun, S.: A probabilistic language based upon sampling functions. In: POPL 2005 (2005)
31. Ramsey, N., Pfeffer, A.: Stochastic lambda calculus and monads of probability distributions. In: POPL 2002 (2002)
32. Reed, J., Pierce, B.C.: Distance makes the types grow stronger: a calculus for differential privacy. In: ICFP 2010 (2010)
33. Rinard, M.: Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In: ICS 2006 (2006)
34. Rinard, M.: Using early phase termination to eliminate load imbalances at barrier synchronization points. In: OOPSLA 2007 (2007)
35. Rinard, M., Hoffmann, H., Misailovic, S., Sidiroglou, S.: Patterns and statistical analysis for understanding reduced resource computing. In: Onward! 2010 (2010)
36. Saheb-Djahromi, N.: Probabilistic LCF. In: MFCS 1978 (1978)

37. Sampson, A., Dietl, W., Fortuna, E., Gnanapragasam, D., Ceze, L., Grossman, D.: Enerj: Approximate data types for safe and general low-power computation. In: PLDI 2011 (2011)
38. Sidiroglou, S., Misailovic, S., Hoffmann, H., Rinard, M.: Managing Performance vs. Accuracy Trade-offs With Loop Perforation. In: FSE 2011 (2011)
39. Smith, M.: Probabilistic abstract interpretation of imperative programs using truncated normal distributions. Electronic Notes in Theoretical Computer Science (2008)
40. Sorber, J., Kostadinov, A., Garber, M., Brennan, M., Corner, M.D., Berger, E.D.: Eon: a language and runtime system for perpetual systems. In: SenSys 2007 (2007)
41. Springer, M.: The algebra of random variables. John Wiley & Sons, Chichester (1979)

Probabilistic Abstractions with Arbitrary Domains

Javier Esparza and Andreas Gaiser

Fakultät für Informatik, Technische Universität München, Germany
{esparza,gaiser}@model.in.tum.de

Abstract. Recent work by Hermanns et al. and Kattenbelt et al. has extended counterexample-guided abstraction refinement (CEGAR) to probabilistic programs. These approaches are limited to predicate abstraction. We present a novel technique, based on the abstract reachability tree recently introduced by Gulavani et al., that can use arbitrary abstract domains and widening operators (in the sense of Abstract Interpretation). We show how suitable widening operators can deduce loop invariants difficult to find for predicate abstraction, and propose refinement techniques.

1 Introduction

Abstraction techniques are crucial for the automatic verification of systems with a finite but very large or infinite state space. The Abstract Interpretation framework provides the mathematical basis of abstraction [8]. Recent work has extended abstraction techniques to probabilistic systems using games [13,14,16,22,23]. The systems (e.g. probabilistic programs) are given semantics in terms of Markov Decision Processes (MDPs), which can model nondeterminism and (using interleaving semantics) concurrency. The key idea is to abstract the MDP into a stochastic 2-Player game, distinguishing between nondeterminism inherent to the system (modeled by the choices of Player 1) and nondeterminism introduced by the abstraction (modeled by Player 2). The construction ensures that the probability of reaching a goal state in the MDP using an optimal strategy is bounded from above and from below by the supremum and infimum of the probabilities of reaching the goal in the 2-Player game when Player 1 plays according to an optimal strategy (and Player 2 is free to play in any way)[1]. An analogous result holds for the probability of reaching a goal state in the MDP using a pessimal strategy.

The abstraction technique of [16,23] and the related [13,14,22] relies on predicate abstraction: an abstract state is an equivalence class of concrete states, where two concrete states are equivalent if they satisfy the same subset of a given set of predicates. The concretization of two distinct abstract states is always disjoint (the *disjointness property*). If the upper and lower bounds obtained using a set of predicates are not close enough, the abstraction is refined by adding new

[1] In [16], the roles of the players are interchanged.

E. Yahav (Ed.): SAS 2011, LNCS 6887, pp. 334–350, 2011.

predicates with the help of interpolation, analogously to the well-known CEGAR approach for non-probabilistic systems.

While predicate abstraction has proved very successful, it is known to have a number of shortcomings: potentially expensive equality and inclusion checks for abstract states, and "predicate explosion". In the non-probabilistic case, the work of Gulavani *et al.* has extended the CEGAR approach to a broader range of abstract domains [12], in which widening operations can be combined with interpolation methods, leading to more efficient abstraction algorithms. We show that the ideas of Gulavani *et al.* can also be applied to probabilistic systems, which extends the approaches of [16,17,23] to arbitrary abstract domains. Given a probabilistic program, an abstract domain and a widening for this domain, we show how to construct an abstract stochastic 2-Player reachability game. The disjointness property is not required. We prove that bounds on the probability of reaching a goal state in the MDP can be computed as in [16,23]. The proofs of [23] use the disjointness property to easily define a Galois connection between the sets of functions assigning values to the abstract and the concrete states. Since there seems to be no easy way to adapt them or the ones from [16,17] to our construction, we show the soundness of our approach by a new proof that uses different techniques.

We also propose an abstraction refinement technique that adapts the idea of delaying the application of widenings [5] to the probabilistic case. The technique delays widenings at the nodes which are likely to have a larger impact in improving the bounds. We present experimental results on several examples.

The paper is organized as follows. In the rest of the introduction we discuss related work and informally present the key ideas of our approach by means of examples. Section 2 contains preliminaries. Section 3 formally introduces the abstraction technique, and proves that games we are considering indeed give us upper resp. lower bound of the exact minimal and maximal reachability probabilities of reaching a set of states. Section 4 shows methods of refining our abstractions and discusses some experiments.

Related work. Besides [13,14,16,22,23], Monniaux has studied in [18] how to abstract probability distributions over program states (instead of the states themselves), but only considers upper bounds for probabilities, as already pointed out in [23]. In [19], Monniaux analyses different quantitative properties of Markov Decision processes, again using abstractions of probability distributions. In contrast, our approach constructs an abstraction using "non-probabilistic" domains and widenings and then performs the computation of strategies and strategy values, which might be used for a refinement of the abstraction. Finally, in [20] Hankin, Di Pierro, and Wiklicky develop a framework for probabilistic Abstract Interpretation which, loosely speaking, replaces abstract domains by linear spaces and Galois connections by special linear maps, and aims at computing expected values of random variables. In contrast, we stick to the standard framework, since in particular we wish to apply existing tools, and aim for upper and lower bounds of probabilities.

1.1 An Example

Consider the following program, written in pseudo code:

```
   int nrp = 0;
1: while (nrp < 100)
2:    if (rec_pack()) then nrp = nrp+1 else break
3: if (nrp < 1) then (fail or goto 1) else exit
```

where the choice between `fail` and `goto 1` is decided by the environment. The goal of the program is to receive up to 100 packets through a network connection. Whenever `rec_pack()` is executed, the connection can break down with probability 0.01, in which case `rec_pack()` returns false and no further packets can be received. If at least one packet is received (line 3) the program terminates[2]; otherwise, the program tries to repair the connection, which may fail or succeed, in which case the program is started again. The choice between success and failure is nondeterministic.

We formalize the pseudo code as a *Nondeterministic Probabilistic Program*, abbreviated NPP[3] (see Fig. 1). A NPP is a collection of guarded commands. A guarded command consists of a name (e.g. `A1`), followed by a guard (e.g. `(ctr = 1) & (nrp < 100)`) and a sequence of pairs of probabilities and update commands (e.g. `0.99: (nrp' = nrp+1)`), separated by '+'. A `reach`-line at the end of the NPP describes the set of states for which we want to compute the reachability probability. We call them the *final* states. In our example, reaching `fail` corresponds to satisfying `(ctr = 3) && (nrp < 1)`. A program execution starts with the initial configuration given by the variable declarations. The program chooses a guarded command whose guard is enabled (i.e., satisfied) by the current state of the program, and selects one of its update commands at random, according to the annotated probabilities. After performing the update, the process is repeated until the program reaches a final state.

The probability of reaching `fail` depends on the behaviour of the environment. The smallest (largest) probability clearly corresponds to the environment always choosing `goto 1` (`fail`), and its value is 0 (0.01). However, a brute force automatic technique will construct the standard semantics of the program, a Markov decision process (MDP) with over 400 states, part of which is shown in Fig. 1. We informally introduce our abstraction technique, which does better and is able to infer tight intervals for both the smallest and the largest probability. It is based on the parallel or menu-based predicate abstraction of [13,22], which we adapt to arbitrary abstract domains.

1.2 Constructing a Valid Abstraction

Given an abstract domain and a widening operator, we abstract the MDP of the program into four different *stochastic 2-Player games* sharing the same arena and

[2] It would be more realistic to set another bound, like 20 packets, but with one packet the probabilities are easy to compute.

[3] NPPs roughly correspond to a subset of the input language of the model checker PRISM [1].

```
  int nrp = 0, ctr = 1;
A1: (ctr = 1) & (nrp < 100)
     -> 0.99:(nrp' = nrp+1)
      + 0.01:(ctr' = 2);
A2: (ctr = 1) & (nrp >= 100)
     -> 1:(ctr' = 3);
A3: (ctr = 2) & (nrp < 1)
     -> 1:(ctr' = 1);
A4: (ctr = 2)
     -> 1:(ctr' = 3);
A5: (ctr = 3) & (nrp >= 1)
     -> 1:(ctr' = 3);
reach: (ctr = 3) & (nrp < 1)
```

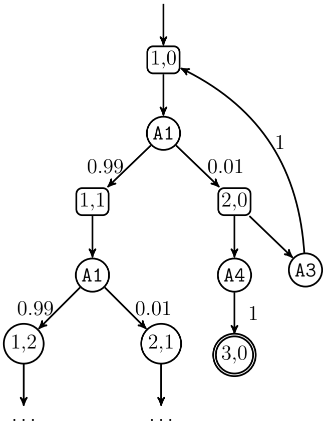


Fig. 1. Example program and prefix of the corresponding Markov Decision Process. Actions are drawn as circles, program states $\langle ctr, nrp\rangle$ as rectangles. $\langle 3, 0\rangle$ is a final state.

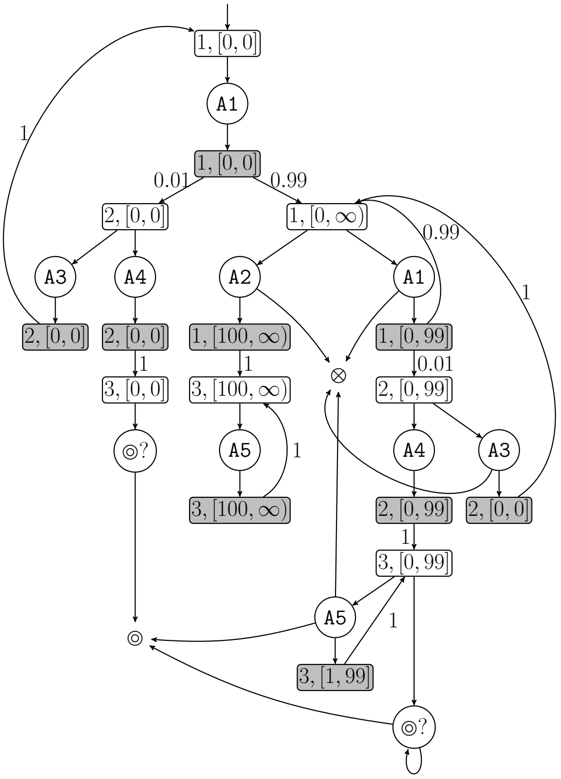


Fig. 2. Abstraction of the program from Fig. 1

the same rules (i.e., the games differ only on the winning conditions). A round of the game starts at an abstract state n (we think of n as a set of concrete states) with Player 1 to move. Let n_i be the set of concrete states of n that enable command A_i. Player 1 proposes an A_i such that $n_i \neq \emptyset$, modeled by a move from n to a node $\langle n, A_i \rangle$. If n contains some final state, then Player 1 can also propose to end the play (modeled by a move to $\langle n, \circledcirc \rangle$). Then it is Player 2's turn. If Player 1 proposes A_i, then Player 2 can accept the proposal (modeled by a move to a node determined below), or reject it and end the play (modeled by a move to another distinguished node $\otimes$), but only if $n_i \neq n$. If Player 2 accepts A_i, she moves to some node $\langle n, A_i, n' \rangle$ such that $n_i \subseteq n'$ (i.e., Player 2 can "pick" a subset n' of n_i out of the subsets offered by the game arena). The next node of the play is determined probabilistically: one of the updates of A_i is selected randomly according to the probabilities, and the play moves to the abstract state obtained by applying the update and (in certain situations) the widening operator to n'. If Player 1 proposes $\circledcirc$ by choosing $\langle n, \circledcirc \rangle$, then Player 2 can accept the proposal, (modeled by a move to $\circledcirc$) or, if not all concrete states of n are final, reject it (modeled by a move $\langle n, \circledcirc \rangle \to \langle n, \circledcirc \rangle$).

Fig. 2 shows an arena for the program of Fig. 1 with the abstract domain and widening operator described in the following. Nodes owned by Player 1 are drawn as white rectangles, nodes owned by Player 2 as circles, and probabilistic nodes as grey rectangles. In the figure we label a node $\langle n, A_i \rangle$ belonging to Player 2 with A_i and a probabilistic node $\langle n, A_i, n' \rangle$ with n' (n resp. n and A_i can easily be reconstructed by inspecting the direct predecessors). Nodes of the form $\langle n, \circledcirc \rangle$ are labeled with '$\circledcirc$?'.

A (concrete) state of the example program is a pair $\langle ctr, nrp \rangle$, and an abstract state is a pair $\langle ctr, [a,b] \rangle$, where $[a,b]$ is an interval of values of `nrp` (i.e., `ctr` is not abstracted in the example). The widening operator ∇ works as follows: if the abstract state $\langle ctr, [a,b] \rangle$ has an ancestor $\langle ctr, [a',b'] \rangle$ along the path between it and the initial state given by the construction, then we overapproximate $\langle ctr, [a,b] \rangle$ by $\langle ctr, s \rangle$, with $s = [a',b'] \nabla [\min(a,a'), \max(b,b')]$. For instance the node $n = \langle 1, [0,\infty) \rangle$ in Fig. 2 enables `A1` and `A2`, and so it has two successors. Since n contains concrete states that do not enable `A1` and concrete states that do not enable `A2`, both of them have $\otimes$ as successor. The node $\langle n, \mathtt{A1}, \langle 1, [0,99] \rangle \rangle$ (whose label is abbreviated by $\langle 1, [0,99] \rangle$ in the figure) is probabilistic. Without widening, its successors would be $\langle 2, [0,99] \rangle$ and $\langle 1, [1,100] \rangle$ with probabilities 0.01 and 0.99. However, $\langle 1, [1,100] \rangle$ has $\langle 1, [0,\infty) \rangle$ as (direct) predecessor, which has the same *ctr*-value. Therefore the widening overapproximates $\langle 1, [1,100] \rangle$ to $\langle 1, [0,\infty) \nabla [0,\infty) \rangle = \langle 1, [0,\infty) \rangle$, and hence we insert an edge from $\langle n, \mathtt{A1}, \langle 1, [0,99] \rangle \rangle$ (back) to $\langle 1, [0,\infty) \rangle$, labeled by 0.99.

After building the arena, we compute lower and upper bounds for the minimal and maximal reachability probabilities as the *values* of four different games, defined as the winning probability of Player 1 for optimal play. The winning conditions of the games are as follows:

(a) Lower bound for the maximal probability: Player 1 wins if the play ends by reaching $\circledcirc$, otherwise Player 2 wins.

```
   int c = 0, i = 0;
1: if choice(0.5) then
2:   while (i <= 100)
3:     i = i+1;
4:     c = c-i+2
5: if (c >= i) then fail
```

```
   int c = 0, i = 0;
1: while(i <= 100)
2:   if choice(0.5) then i = (i+1);
3:   c = c-i+2;
4: if (c >= i) then fail
```

Fig. 3. Example programs 2 and 3

(b) Upper bound for the maximal probability: Players 1 and 2 both win if the play ends by reaching $\circledcirc$, and both lose otherwise.
(c) Lower bound for the minimal probability: Players 1 and 2 both lose if the play ends by reaching $\circledcirc$ or $\otimes$, and both win otherwise.
(d) Upper bound for the minimal probability: Player 1 loses if the play ends by reaching $\circledcirc$ or $\otimes$, otherwise Player 2 loses.

For the intuition behind these winning conditions, consider first game (a). Since Player 1 models the environment and wins by reaching $\circledcirc$, the environment's goal is to reach a final state. Imagine first that the abstraction is the trivial one, i.e., abstract and concrete states coincide. Then Player 2 never has a choice, and the optimal strategy for Player 1 determines a set S of action sequences whose total probability is equal to the maximal probability of reaching a final state. Imagine now that the abstraction is coarser. In the arena for the abstract game the sequences of S are still possible, but now Player 2 may be able to prevent them, for instance by moving to $\otimes$ when an abstract state contains concrete states not enabling the next action in the sequence. Therefore, in the abstract game the probability that Player 1 wins can be at most equal to the maximal probability. In game (b) the team formed by the two players can exploit the spurious paths introduced by the abstraction to find a strategy leading to a better set of paths; in any case, the probability of S is a lower bound for the winning probability of the team. The intuition behind games (c) and (d) is similar.

In our example, optimal strategies for game (b) are: for Player 1, always play the "rightmost" choice, except at $\langle 2, [0, 99] \rangle$, where she should play `A4`; for Player 2, play $\circledcirc$ if possible, otherwise anything but $\otimes$. The value of the game is 1. In game (a), the optimal strategy for Player 1 is the same, whereas Player 2 always plays $\otimes$ (resp. stays in $\circledcirc$?) whenever possible. The value of the game is 0.01. We get $[0.01, 1]$ as lower and upper bound for the maximal probability. For the minimal probability we get the trivial bounds $[0, 1]$.

To get more precision, we can skip widenings at certain situations during the construction. If we e.g. apply widening only after the second unrolling of the loop, the resulting abstraction allows us to obtain the more precise bounds $[0, 0.01]$ and $[0.01, 0.01]$ for minimal and maximal reachability, respectively.

The main theoretical result of our paper is the counterpart of the results of [16,13]: for arbitrary abstraction domains, the values of the four games described above indeed yield upper and lower bounds of the maximal and minimal probability of reaching the goal nodes.

In order to give a first impression of the advantages of abstraction domains beyond predicate abstraction in the probabilistic case, consider the (deterministic) pseudo code on the left of Fig. 3, a variant of the program above. Here `choice`(p) models a call to a random number generator that returns 1 with probability p and 0 with probability $1 - p$.

It is easy to see that $c \leq 1$ is a global invariant, and so the probability of failure is exactly 0.5. Hence a simple invariant like $c \leq k$ for a $k \leq 100$, together with the postcondition $i > 100$ of the loop would be sufficient to negate the guard of the statement at line 5. However, when this program is analysed with PASS [13,14], a leading tool on probabilistic abstraction refinement on the basis of predicate abstraction, the while loop is unrolled 100 times because the tool fails to "catch" the invariant, independently of the options chosen to refine the abstraction [4].

On the other hand, an analysis of the program with the standard interval domain, the standard widening operator, and the standard technique of delaying widenings [5], easily 'catches" the invariant (see Section 4.1). The same happens for the program on the right of the figure, which exhibits a more interesting probabilistic behaviour, especially a probabilistic choice within a loop: we obtain good upper and lower bounds for the probability of failure using the standard interval domain. Notice that examples exhibiting the opposite behaviour (predicate abstraction succeeds where interval analysis fails) are not difficult to find; our thesis is only that the game-based abstraction approach of [13,16] can be extended to arbitrary abstract domains, making it more flexible and efficient.

2 Stochastic 2-Player Games

This section introduces stochastic 2-Player games. For a more thorough introduction into the subject and proofs for the theorems see e.g. [21,6,7].

Let S be a countable set. We denote by $\mathrm{Dist}(S)$ the set of all distributions $\delta : S \to [0, 1]$ over S with $\delta(x) = 0$ for all but finitely many $x \in S$.

Definition 1. *A* stochastic 2-Player game $\mathcal{G}$ *(short 2-Player game) is a tuple* $((V_1, V_2, V_p), E, \delta, s_0)$*, where*

- V_1, V_2, V_p *are distinct, countable sets of states. We set* $V = V_1 \cup V_2 \cup V_p$*;*
- $E \subseteq (V_1 \cup V_2) \times V$ *is the set of* admissible *player choices;*
- $\delta : V_p \to \mathit{Dist}(V)$ *is a probabilistic transition function;*
- $s_0 \in V_1$ *is the start state.*

Instead of $(q, r) \in E$ *we often write* $q \to r$*. A string* $w \in V^+$ *is a* finite run *(short: run) of* $\mathcal{G}$ *if (a)* $w = s_0$*, or (b)* $w = w's's$ *for some run* $w's' \in V^*(V_1 \cup V_2)$ *and* $s' \to s$*, or (c)* $w = w's's$ *for some run* $w's' \in V^*V_p$ *such that* $\delta(s')(s) > 0$*. We denote the set of all runs of* $\mathcal{G}$ *by* $\mathit{Cyl}(\mathcal{G})$*. A run* $w = x_1 \ldots x_k$ *is* accepting

[4] Actually, the input language of PASS does not explicitly include while loops, they have to be simulated. But this does not affect the analysis.

relative to $F \subseteq V_1$ *if* $x_k \in F$ *and* $x_i \notin F$ *for* $1 \leq i < k$. *The set of accepting runs relative to* F *is denoted by* $Cyl(\mathcal{G}, F)$.

A stochastic 2-Player game with $V_2 = \emptyset$ *is called a* Markov Decision Process (MDP)*, and then we write* $\mathcal{G} = ((V_1, V_p), E, \delta, s_0)$.

Fix for the rest of the section a 2-Player game $\mathcal{G} = ((V_1, V_2, V_p), E, \delta, s_0)$. The behaviours of Player 1 and 2 in $\mathcal{G}$ are described with the help of *strategies*:

Definition 2. *A strategy for Player* $i \in \{1, 2\}$ *in* $\mathcal{G}$ *is a partial function* $\phi : Cyl(\mathcal{G}) \to Dist(V)$ *satisfying the following two conditions:*

- $\phi(w)$ *is defined iff* $w = w'v \in V^*V_i$ *and* $v \to x$ *for some* $x \in V$*; and*
- *if* $\phi(w)$ *is defined and* $\phi(w)(x) > 0$ *then* wx *is a run.*

We denote the set of strategies for Player i by $S_i(\mathcal{G})$. *A strategy* ϕ *is* memoryless *if* $\phi(w_1) = \phi(w_2)$ *for any two runs* w_1, w_2 *ending in the same node of* V_i*, and* non-randomized *if for every run* w *such that* $\phi(w)$ *is defined there is a node* x *such that* $\phi(w)(x) = 1$. *Given strategies* ϕ_1, ϕ_2 *for Players 1 and 2, the* value $val(w)_{\mathcal{G}[\phi_1,\phi_2]}$ *of a run* w *under* ϕ_1, ϕ_2 *is defined as follows:*

- *If* $w = s_0$*, then* $val(w)_{\mathcal{G}[\phi_1,\phi_2]} = 1$.
- *If* $w = w's \in V^*V_i$ *for* $i \in \{1, 2\}$ *and* $\phi_i(w')$ *is defined, then* $val(w)_{\mathcal{G}[\phi_1,\phi_2]} = val(w')_{\mathcal{G}[\phi_1,\phi_2]} \cdot \phi_i(w')(s)$.
- *If* $w = w's's$ *for some run* $w's' \in V^*V_p$ *then* $val(w)_{\mathcal{G}[\phi_1,\phi_2]} = val(w's')_{\mathcal{G}[\phi_1,\phi_2]} \cdot \delta(s')(s)$.
- *Otherwise* $val(w)_{\mathcal{G}[\phi_1,\phi_2]} = 0$.

We are interested in *probabilistic reachability*:

Definition 3. *The* probability $Reach(\mathcal{G}[\phi_1, \phi_2], F)$ of reaching $F \subset V_1$ *in* $\mathcal{G}$ *under strategies* ϕ_1 *and* ϕ_2 *of Players 1 and 2 is*

$$Reach(\mathcal{G}[\phi_1, \phi_2], F) := \sum_{w \in Cyl(\mathcal{G},F)} val(w)_{\mathcal{G}[\phi_1,\phi_2]}.$$

If the context is clear, we often omit the subscript of val$(\cdot)$. We write Cyl$(\mathcal{G}[\phi_1, \phi_2])$ (resp. Cyl$(\mathcal{G}[\phi_1, \phi_2], F)$) for the set of all finite runs $r \in$ Cyl$(\mathcal{G})$ (resp. $r \in$ Cyl$(\mathcal{G}, F)$) with val$(r)_{\mathcal{G}[\phi_1,\phi_2]} > 0$. In a MDP $\mathcal{M}$ we do not require to have a strategy for the second Player. Here we just write Reach$(\mathcal{M}[\phi_1], F)$ for a given strategy $\phi_1 \in S_1(\mathcal{M})$.

Definition 4. *Let* $\mathcal{G} = ((V_1, V_2, V_p), E, \delta, s_0)$ *be a 2-Player game, and* $F \subset V_1$. *The* extremal game values $Reach(\mathcal{G}, F)^{++}, Reach(\mathcal{G}, F)^{+-}, Reach(\mathcal{G}, F)^{-+}$ *and* $Reach(\mathcal{G}, F)^{--}$ *are*

$$\begin{aligned}
Reach(\mathcal{G}, F)^{++} &:= \sup_{\phi_1 \in S_1(\mathcal{G})} \sup_{\phi_2 \in S_2(\mathcal{G})} Reach(\mathcal{G}[\phi_1, \phi_2], F)\\
Reach(\mathcal{G}, F)^{+-} &:= \sup_{\phi_1 \in S_1(\mathcal{G})} \inf_{\phi_2 \in S_2(\mathcal{G})} Reach(\mathcal{G}[\phi_1, \phi_2], F)\\
Reach(\mathcal{G}, F)^{-+} &:= \inf_{\phi_1 \in S_1(\mathcal{G})} \sup_{\phi_2 \in S_2(\mathcal{G})} Reach(\mathcal{G}[\phi_1, \phi_2], F)\\
Reach(\mathcal{G}, F)^{--} &:= \inf_{\phi_1 \in S_1(\mathcal{G})} \inf_{\phi_2 \in S_2(\mathcal{G})} Reach(\mathcal{G}[\phi_1, \phi_2], F)
\end{aligned}$$

If $\mathcal{G}$ is a MDP, we define $Reach(\mathcal{G}, F)^+ := Reach(\mathcal{G}, F)^{++} = Reach(\mathcal{G}, F)^{+-}$ and $Reach(\mathcal{G}, F)^- := Reach(\mathcal{G}, F)^{--} = Reach(\mathcal{G}, F)^{-+}$.

The following well-known theorem will be crucial for the validity of our abstractions [6]:

Theorem 1. *Let $F \subset V_1$. For each $\kappa \in \{++, +-, -+, --\}$ there exist* non-randomized and memoryless *strategies $\phi_1^\kappa \in S_1(\mathcal{G}), \phi_2^\kappa \in S_2(\mathcal{G})$ such that*

$$Reach(\mathcal{G}, F)^\kappa = Reach(\mathcal{G}[\phi_1^\kappa, \phi_2^\kappa], F).$$

Extremal game values can be computed e.g. by variants of value iteration [7].

3 Abstractions of Probabilistic Programs

We start by giving a formal definition of NPPs.

Definition 5. *Let $\mathcal{V}$ be a finite set of variables, where $x \in \mathcal{V}$ has a range $rng(x)$. A* configuration *(or state) of $\mathcal{V}$ is a map $\sigma \colon \mathcal{V} \to \bigcup_{x \in \mathcal{V}} rng(x)$ such that $\sigma(x) \in rng(x)$ for all $x \in \mathcal{V}$. The set of all configurations is denoted by $\Sigma_{\mathcal{V}}$. A* transition *is a map $f \in 2^{\Sigma_{\mathcal{V}}} \to 2^{\Sigma_{\mathcal{V}}}$ such that $|f(\{\sigma\})| \leq 1$ for all $\sigma \in \Sigma_{\mathcal{V}}$ (i.e., a transition maps a single configuration to the empty set or to a singleton again), and*

$$\bigcup_{\sigma \in M} f(\{\sigma\}) = f(M) \text{ for all } M \subseteq \Sigma_{\mathcal{V}}.$$

A transition g is a guard *if $g(\{\sigma\}) \in \{\{\sigma\}, \emptyset\}$ for every configuration σ. We say that σ* enables *g if $g(\{\sigma\}) = \{\sigma\}$. A transition c is an* assignment *if $|c(\{\sigma\})| = 1$ for all $\sigma \in \Sigma_{\mathcal{V}}$. The semantics of an assignment c is the map $[\![c]\!] : \Sigma_{\mathcal{V}} \to \Sigma_{\mathcal{V}}$ given by $[\![c]\!](\sigma) := \sigma'$ if $c(\{\sigma\}) = \{\sigma'\}$. The set of transitions is denoted by $Trans_{\mathcal{V}}$.*

Definition 6. *Nondeterministic Probabilistic Programs.*
A nondeterministic probabilistic program *(NPP) is a triple $P = (\mathcal{V}, \sigma_0, \mathcal{C})$ where $\mathcal{V}$ is a finite set of program* variables*, $\sigma_0 \in \Sigma_{\mathcal{V}}$ is the* initial configuration*, and $\mathcal{C}$ is a finite set of* guarded commands*. A* guarded command *A has the form $A = g \to p_1 : c_1 + \ldots + p_m : c_m$, where $m \geq 1$, g is a guard, $p_1, \ldots, p_m$ are probabilities adding up to 1, and $c_1, \ldots, c_m$ are assignments. We denote the guard of A by g_A, the* updates *$\{\langle p_1, c_1 \rangle, \ldots \langle p_m, c_m \rangle\}$ of A by up_A, and the set $\{up_A \mid A \in \mathcal{C}\}$ by $up_{\mathcal{C}}$.*

Definition 7. *Semantics of NPPs and Reachability Problem.*
The MDP *associated to a NPP $P = (\mathcal{V}, \sigma_0, \mathcal{C})$ is $\mathcal{M}_P = ((V_1, V_p), E, \delta, \sigma_0)$, where $V_1 = \Sigma_{\mathcal{V}}$, $V_p = \Sigma_{\mathcal{V}} \times \mathcal{C}$, $E \subseteq V_1 \times (V_1 \cup V_p)$, $\delta : (\Sigma_{\mathcal{V}} \times \mathcal{C}) \to Dist(V_1)$, and for every $A \in \mathcal{C}$, $\sigma, \sigma' \in \Sigma_{\mathcal{V}}$*

$$\sigma \to \langle \sigma, A \rangle \text{ iff } \sigma \text{ enables } g_A \text{ and } \delta(\langle \sigma, A \rangle)(\sigma') := \sum_{\langle p,c \rangle \in up_A \colon [\![c]\!](\sigma) = \sigma'} p\ .$$

The reachability problem *for P relative to a set $F \subseteq \Sigma_{\mathcal{V}}$ of states such that $\sigma_0 \notin F$ is the problem of computing $Reach(\mathcal{M}_P, F)^+$ and $Reach(\mathcal{M}_P, F)^-$. We call F the set of* final states.

We assume in the following that for every run $w\sigma \in V^*V_1$ in $\mathcal{M}_P$ either $\sigma \in F$ or σ enables the guard of at least one command (i.e., we do not 'get stuck' during the computation). This can e.g. be achieved by adding a suitable guarded command that simulates a self loop.

3.1 Abstracting NPPs

We abstract NPPs using the Abstract Interpretation framework (see [8]). As usual, an abstract domain is a complete lattice $(D^\sharp, \sqsubseteq, \top, \bot, \sqcup, \sqcap)$ (short $D^\sharp$), and we assume the existence of monotone abstraction and concretization maps $\alpha : 2^{\Sigma_\mathcal{V}} \to D^\sharp$ and $\gamma : D^\sharp \to 2^{\Sigma_\mathcal{V}}$ forming a Galois connection between $D^\sharp$ and 2^{Σ_V}. A *widen operator* is a mapping $\nabla : D^\sharp \times D^\sharp \to D^\sharp$ satisfying (i) $a \nabla b \sqsupseteq a$ and $a \nabla b \sqsupseteq b$ for all $a, b \in D^\sharp$, and (ii) for every strictly increasing sequence $a_0 \sqsubset a_1 \sqsubset \ldots$ in $D^\sharp$ the sequence $(b_i)^{i \in \mathbb{N}}$ defined by $b_0 = a_0$ and $b_{i+1} = b_i \nabla a_{i+1}$ is stationary.

We abstract sets of configurations by elements of $D^\sharp$. Following ideas from [16,13,14,22], the abstraction of an NPP is a 2-player stochastic game. We formalize which games are *valid abstractions* of a given NPP (compare the definition to the comments in Section 1.2):

Definition 8. *Let $P = (\mathcal{V}, \sigma_0, \mathcal{C})$ be a NPP with a set $F \subseteq \Sigma_\mathcal{V}$ of final states such that $\sigma_0 \notin F$. A 2-player game $\mathcal{G} = ((V_1, V_2, V_p), E, \delta, s_0)$ with finitely many nodes is a* valid abstraction *of P relative to F for $D^\sharp$ if*

- *V_1 contains a subset of $D^\sharp$ plus two distinguished states $\circledcirc, \otimes$;*
- *V_2 is a set of pairs $\langle s, A \rangle$, where $s \in V_1 \setminus \{\circledcirc, \otimes\}$ and either $A = \circledcirc$ or A is a command of $\mathcal{C}$ enabled by some state of $\gamma(s)$;*
- *V_p is a set of fourtuples $\langle s, A, s', d \rangle$, where $s, s' \in V_1 \setminus \{\circledcirc, \otimes\}$ such that $s \sqsupseteq s'$, A is a command enabled by some state of $\gamma(s')$, and d is the mapping that assigns to every update $\langle p, c \rangle \in up_A$ an abstract state $s' \in V_1$ with $\gamma(d(\langle p, c \rangle)) \supseteq c(\gamma(s'))$;*
- *$s_0 = \alpha(\{\sigma_0\})$;*

and the following conditions hold:

1. *For every $s \in V_1 \setminus \{\circledcirc, \otimes\}$ and every $A \in \mathcal{C}$:*
   (a) *If $\gamma(s) \cap F \neq \emptyset$ then $s \to \langle s, \circledcirc \rangle \to \circledcirc$. If moreover $\gamma(s) \subseteq F$, then $\langle s, \circledcirc \rangle$ is the only successor of s; otherwise, also $\langle s, \circledcirc \rangle \to \langle s, \circledcirc \rangle$ holds.*
   (If $\gamma(s)$ contains some final state, then Player 1 can propose $\circledcirc$. If all states of $\gamma(s)$ are final, then Player 2 must accept, otherwise it can accept, or reject by staying in $\langle s, \circledcirc \rangle$. *)*
   (b) *If $g_A(\gamma(s)) \neq \emptyset$ then $\langle s, A \rangle \in V_2$ and $s \to \langle s, A \rangle$.*
   (If some state of $\gamma(s)$ enables A then Player 1 can propose A. *)*
2. *For every pair $\langle s, A \rangle \in V_2$ and every $A \in \mathcal{C}$:*
   (a) *there exist nodes $\{\langle s, A, s_1, d_1 \rangle, \ldots, \langle s, A, s_k, d_k \rangle\} \subseteq V_p$ such that $\langle s, A \rangle \to \langle s, A, s_i, d_i \rangle$ for every $i \leq k$ and $g_A(\gamma(s)) \subseteq \bigcup_{j=1}^{k} \gamma(s_j)$.*
   (If Player 2 accepts A, then she can pick any concrete state $\sigma \in \gamma(s)$ enabling A, and choose a successor $\langle s, A, s_i, d_i \rangle$ such that $\sigma \in \gamma(s_i)$. *)*

(b) If $\gamma(s) \cap F \neq \emptyset$, then $\langle s, A\rangle \to \circledcirc$.
(If $\gamma(s)$ contains some final state, then Player 2 can reject and move to $\circledcirc$. *)*

(c) If $g_A(\gamma(s)) \neq \gamma(s)$, then $\langle s, A\rangle \to \otimes$.
(If some state of $\gamma(s)$ does not enable A, then Player 2 can reject A and move to $\otimes$. *)*

3. For every $\langle s, A, s', d\rangle \in V_p$ and every abstract state $s'' \in V_1$:

$$\delta(\langle s, A, s', d\rangle)(s'') := \sum_{\langle p,c\rangle \in up_A :\ d(\langle p,c\rangle)=s''} p\ .$$

4. The states $\otimes$ and $\circledcirc$ have no outgoing edges.

We can now state the main theorem of the paper: the extremal game values of the games derived from valid abstractions provide upper and lower bounds on the maximal and minimal reachability probabilities. The complete proof is given in [10].

Theorem 2. *Let P be a NPP and let $\mathcal{G}$ be a valid abstraction of P relative to F for the abstract domain $D^\sharp$. Then*

$$\mathit{Reach}(\mathcal{M}_P, F)^- \in [\mathit{Reach}(\mathcal{G}, \{\circledcirc, \otimes\})^{--}, \mathit{Reach}(\mathcal{G}, \{\circledcirc, \otimes\})^{-+}] \text{ and}$$
$$\mathit{Reach}(\mathcal{M}_P, F)^+ \in [\mathit{Reach}(\mathcal{G}, \{\circledcirc\})^{+-}, \mathit{Reach}(\mathcal{G}, \{\circledcirc\})^{++}].$$

Proof. (*Sketch.*) The result is an easy consequence of the following three assertions:

(1) Given a strategy ϕ of the (single) player in $\mathcal{M}_P$, there exists a strategy $\phi_1 \in S_1(\mathcal{G})$ such that

$$\inf_{\psi \in S_2(\mathcal{G})} \mathrm{Reach}(\mathcal{G}[\phi_1, \psi], \{\circledcirc, \otimes\}) \le \mathrm{Reach}(\mathcal{M}_P[\phi], F) \text{ and}$$

$$\sup_{\psi \in S_2(\mathcal{G})} \mathrm{Reach}(\mathcal{G}[\phi_1, \psi], \{\circledcirc\}) \ge \mathrm{Reach}(\mathcal{M}_P[\phi], F).$$

(2) Given a strategy $\phi_1 \in S_1(\mathcal{G})$ there exists a strategy $\phi \in S_1(\mathcal{M}_P)$ such that

$$\mathrm{Reach}(\mathcal{M}_P[\phi], F) \le \sup_{\psi \in S_2(\mathcal{G})} \mathrm{Reach}(\mathcal{G}[\phi_1, \psi], \{\otimes, \circledcirc\}).$$

(3) Given a strategy $\phi_1 \in S_1(\mathcal{G})$ there exists a strategy $\phi \in S_1(\mathcal{M}_P)$ such that

$$\mathrm{Reach}(\mathcal{M}_P[\phi], F) \ge \inf_{\psi \in S_2(\mathcal{G})} \mathrm{Reach}(\mathcal{G}[\phi_1, \psi], \{\circledcirc\}).$$

To prove (1) (the other two assertions are similar), we use ϕ to define a function D that distributes the probabilistic mass of a run $R \in \mathrm{Cyl}(\mathcal{G})$ among all the runs $r \in \mathrm{Cyl}(\mathcal{M})$ (where $\mathcal{M}$ is a normalization of $\mathcal{M}_P$). The strategies ϕ_1 and ϕ_2 are then chosen so that they produce the same distribution, i.e., the mass of all the runs r that follow the strategies and correspond to an abstract run R following ϕ is equal to the mass of R. □

Recall that in predicate abstraction the concretizations of two abstract states are disjoint sets of configurations (disjointness property). This allows to easily define a Galois connection between the sets of functions assigning values to the abstract and the concrete states: Given a concrete valuator f, its abstraction is the function that assigns to a set X the minimal resp. the maximal value assigned by f to the elements of X. Here we have to distribute the value of a concrete state into multiple abstract states (which is what we do in our proof).

3.2 An Algorithm for Constructing Valid Abstractions

Algorithm 1 builds a valid abstraction $\mathcal{G}$ of a NPP P relative to a set F of final states for a given abstract domain $D^\sharp$. It is inspired by the algorithms of [4,11] for constructing abstract reachability trees. It constructs the initial state $s_0 = \alpha(\{\sigma_0\})$ and generates transitions and successor states in a breadth-first fashion using a work list called work. The GENSUCCS procedure constructs the successors of a node guided by the rules from Def. 8. It uses abstract transformers $g^\sharp$ and $c^\sharp$ for the guards and commands of the NPP. Hereby a transformer $g^\sharp : D^\sharp \to 2^{D^\sharp}$ abstracting a guard g has to satisfy that for all $a \in D^\sharp$, $g^\sharp(a)$ is finite and $\bigcup_{b \in g^\sharp(a)} \gamma(b) \supseteq g(\gamma(a))$. Allowing $g^\sharp$ to return a set rather than just one element from $D^\sharp$ can help increasing the accuracy of $\mathcal{G}$. Here we implicitly make use of abstract powerset domains. GENSUCCS assumes that it can be decided whether $\gamma(s) \cap F = \emptyset$, $\gamma(s) \nsubseteq F$, $g_A(\gamma(s)) \neq \emptyset$ or $g_A(\gamma(s)) \neq \gamma(s)$ hold (lines 2 and 4). The assumptions on F are reasonable, since in most cases the set F has a very simple shape, and could be replaced by conservative tests on the abstract. A conservative decision procedure suffices for the test $g_A(\gamma(s)) \neq \gamma(s)$, with the only requirement that if it returns 0, then $g_A(\gamma(s)) = \gamma(s)$ has to hold. The same holds for the test $g_A(\gamma(s)) \neq \emptyset$. GENSUCCS closely follows the definition of a valid abstraction, as specified in Def. 8.

Lines 1 and 2 guarantee that condition (1a) of Def. 8 holds, and, similarly, line 4 guarantees condition (1b). Similarly, lines 3 and 5 are needed to satisfy conditions (2b) and (2c), respectively. The loop at line 6 generates the nodes of the form $\langle s, A, s_i, d \rangle$ required by condition (2a) of our definition, and the loop at line 7 constructs the function d appearing in condition 3.

As usual, termination of the algorithm requires to use widenings. This is the role of the EXTRAPOLATE procedure. During the construction, we use the function pred$(\cdot)$ to store for every node $s \in V_1 \setminus \{s_0, ⊚, \otimes\}$ its predecessor in the spanning tree induced by the construction (we call it *the spanning tree* from now on). For a node $s' \in V_1$ that was created as the result of chosing a guarded command A, the procedure finds the nearest predecessor s in the spanning tree with the same property, and uses s to perform a widen operation. Note that in the introductory example, another strategy was used: There we applied widenings only for states with matching control location. The strategy used in EXTRAPOLATE does not use additional information like control flow and thus can be used for arbitrary NPPs. We can now prove (see [10]):

Theorem 3. *Algorithm 1 terminates, and its result $\mathcal{G}$ is a valid abstraction.*

Algorithm 1: Computing $\mathcal{G}$.

```
Input: NPP $P = (\mathcal{V}, \sigma_0, \mathcal{C})$, abstract domain $D^\sharp$, set of final states $F \subseteq \Sigma_\mathcal{V}$,
       widening $\nabla$.
Output: 2-Player game $\mathcal{G} = ((V_1, V_2, V_p), E, \delta, s_0)$.

$s_0 = \alpha(\{\sigma_0\})$; pred($s_0$) $\leftarrow$ nil
$V_1 \leftarrow \{s_0, \otimes, \circledcirc\}$; $V_2 \leftarrow \emptyset$; $V_p \leftarrow \emptyset$; work $\leftarrow \{s_0\}$
while $work \neq \emptyset$ do
    Remove $s$ from the head of work; GENSUCCS($s$)

Procedure GENSUCCS($s \in V_1$)
fopt $\leftarrow$ $false$
if $\gamma(s) \cap F \neq \emptyset$ then
    $E \leftarrow E \cup \{(s, \langle s, \circledcirc\rangle), (\langle s, \circledcirc\rangle, \circledcirc)\}$
    if $\gamma(s) \not\subseteq F$ then { $E \leftarrow E \cup \{(\langle s, \circledcirc\rangle, \langle s, \circledcirc\rangle)\}$; fopt $\leftarrow$ $true$ }
    else return
forall the $A \in \mathcal{C}$ do
    if $g_A(\gamma(s)) \neq \emptyset$ then
        $V_2 \leftarrow V_2 \cup \{\langle s, A\rangle\}$; $E \leftarrow E \cup \{(s, \langle s, A\rangle)\}$
        if $g_A(\gamma(s)) \neq \gamma(s)$ then $E \leftarrow E \cup \{(\langle s, A\rangle, \otimes)\}$
        if $fopt$ then $E \leftarrow E \cup \{(\langle s, A\rangle, \circledcirc)\}$
        forall the $s' \in g^\sharp_A(s)$ do
            Create a fresh array $d : up_\mathcal{C} \rightarrow V_1$
            forall the $\langle p, c\rangle \in up_A$ do
                $v \leftarrow$ EXTRAPOLATE($c^\sharp(s), s, A$); $d(\langle p, c\rangle) \leftarrow v$
                if $v \notin V_1$ then {
                $V_1 \leftarrow V_1 \cup \{v\}$; pred($v$) $= \langle s, A\rangle$; add $v$ to work }
            $V_p \leftarrow V_p \cup \{\langle s, A, s', d\rangle\}$; $E \leftarrow E \cup \{(\langle s, A\rangle, \langle s, A, s', d\rangle)\}$

Procedure EXTRAPOLATE($v \in D^\sharp, s \in V_1 \setminus \{\circledcirc, \otimes\}, A \in \mathcal{C}$)
$\langle s', A'\rangle \leftarrow$ pred($s$)
while $pred(s') \neq nil$ do
    if $A' = A$ then return $s \nabla (s \sqcup v)$
    else { buffer $\leftarrow s'$; $\langle s', A'\rangle \leftarrow$ pred($s'$); $s \leftarrow$ buffer }
return $v$
```

4 Refining Abstractions: Quantitative Widening Delay

Algorithm 1 applies the widening operator whenever the current node has a predecessor in the spanning tree that was created by the application applying the same guarded command. This strategy usually leads to too many widenings and poor abstractions. A popular solution in non-probabilistic abstract interpretation is to delay widenings in an initial stage of the analysis [5], in our case until the spanning tree reaches a given depth. We call this approach *depth-based unrolling*.

Note that if $\mathcal{M}_P$ is finite and the application of widenings is the only source of imprecision, this simple refinement method is complete.

A shortcoming of this approach is that it is insensitive to the probabilistic information. We propose to combine it with another heuristic. Given a valid abstraction $\mathcal{G}$, our procedure yields two pairs (ϕ_1^+, ϕ_2^+) resp. (ϕ_1^-, ϕ_2^-) of memoryless and non-probabilistic strategies that satisfy $\text{Reach}(\mathcal{G}[\phi_1^-, \phi_2^-], \{\circledcirc\}) = \text{Reach}(\mathcal{G}, \{\circledcirc\})^{+-}$ resp. $\text{Reach}(\mathcal{G}[\phi_1^+, \phi_2^+], \{\circledcirc\}) = \text{Reach}(\mathcal{G}, \{\circledcirc\})^{++}$. Given a node s for Player 1, let P_s^+ and P_s^- denote the probability of reaching $\circledcirc$ (resp. $\circledcirc$ or $\otimes$ if we are interested in minimal probabilities) starting at s and obeying the strategies (ϕ_1^+, ϕ_2^+) resp. (ϕ_1^-, ϕ_2^-) in $\mathcal{G}$. In order to refine $\mathcal{G}$ we can choose any node $s \in V_1 \cap D^\sharp$ such that $P_s^+ - P_s^- > 0$ (i.e., a node whose probability has not been computed exactly yet), such that at least one of the direct successors of s in the spanning tree has been constructed using a widening. We call these nodes the *candidates* (for delaying widening). The question is which candidates to select. We propose to use the following simple heuristic:

> Sort the candidates s according to the product $w_s \cdot (P_s^+ - P_s^-)$, where w_s denotes the product of the probabilities on the path of the spanning tree of $\mathcal{G}$ leading from s_0 to s. Choose the n candidates with largest product, for a given n.

We call this heuristic the *mass heuristic*. The *mixed heuristic* delays widenings for nodes with depth less than a threshold i, and for n nodes of depth larger than or equal to i with maximal product. In the next section we illustrate depth-based unrolling, the mass heuristic, and the mixed heuristic on some examples.

4.1 Experiments

We have implemented a prototype of our approach on top of the Parma Polyhedra Library [3], which provides several numerical domains [2]. We present some experiments showing how simple domains like intervals can outperform predicate abstraction. Notice that examples exhibiting the opposite behaviour are also easy to find: our experiments are not an argument against predicate abstraction, but an argument for abstraction approaches not limited to it.

If the computed lower and upper bounds differ by more than 0.01, we select refinement candidates using the different heuristics presented before and rebuild the abstraction. We used a Linux machine with 4GB RAM.

Two small programs. Consider the NPPs of Fig. 4. We compute bounds with different domains: intervals, octagons, integer grids, and the product of integer grids and intervals [9]. For the refinement we use the mass (M) depth (D) and mixed (Mix) heuristics. For M and Mix we choose 15 refinement candidates at each iteration. The results are shown in Table 1. For the left program the integer grid domain (and the product) compute precise bounds after one iteration. After 10 minutes, the PASS tool [13] only provides the bounds $[0.5, 0.7]$ for the optimal reachability probability. For the right program only the product of grids and intervals is able to "see" that $x \equiv 0 \pmod 3$ or $y < 30$ holds, and yields

```
int a=0, ctr=0;
A1: (ctr=0)
     -> 0.5:(a’=1)&(ctr’=1)
       +0.5:(a’=0)&(ctr’=1);
A2: (ctr=1)&(a>=-400)&(a<= 400)
     -> 0.5:(a’=a+5)
       +0.5:(a’=a-5);
A3: (ctr=1) -> 1:(ctr’=2);
reach: (a=1)&(ctr=2)
```

```
int x=0, y=0, c=0;
A1: (c=0)&(x<=1000)
     -> 0.25:(x’=3*x+2)&(y’=y-x)
       +0.75:(x’=3*x)&(y’=30);
A2: (c=0)&(x>1000) -> 1:(c’=1);
A3: (c=1)&(x>=3) -> 1:(x’=x-3);
reach: (c=1)&(x=2)&(y>=30)
```

Fig. 4. Two guarded-command programs

precise bounds after 3 refinement steps. After 10 minutes PASS only provides the bounds $[0, 0.75]$. The example illustrates how pure depth-based unrolling, ignoring probabilistic information, leads to poor results: the mass and mixed heuristics perform better. PASS may perform better after integrating appropriate theories, but the example shows that combining domains is powerful and easily realizable by using Abstract Interpretation tools.

Programs of Fig. 3. For these PASS does not terminate after 10 minutes, while with the interval domain our approach computes the exact value after at most 5 iterations and less than 10 seconds. Most of the predicates added by PASS during the refinement for program 2 have the form $\mathtt{c} \leq \alpha \cdot \mathtt{i} + \beta$ with $\alpha > 0, \beta < 0$: PASS's interpolation engines seem to take the wrong guesses during the generation of new predicates. This effect remains also if we change the refinement strategies of PASS. PASS offers the option of manually adding predicates. Interestingly it suffices to add a predicate as simple as e.g. $i > 3$ to help the tool deriving the solution after 3 refinements for program 2.

Zeroconf. This is a simple probabilistic model of the Zeroconf protocol, adapted from [1,16], where it was analyzed using PRISM and predicate abstraction. It is parameterized by K, the maximal number of probes sent by the protocol. We check it for $K = 4, 6, 8$ and two different properties. *Zeroconf* is a very good example for predicate abstraction, and so it is not surprising that PASS beats the interval domain (see Table 2). The example shows how the mass heuristic by

Table 1. Experimental results for the programs in Fig. 4. Iters is the number of iterations needed. Time is given in seconds. '-' means the analysis did not return a precise enough bound after 10 minutes. Size denotes the maximal number of nodes belonging to Player 1 that occured in one of the constructed games.

Program	Value	Interval			Octagon			Grid			Product		
		M	D	Mix	M	D	Mix	M	D	Mix	M	D	Mix
Left	Iters:	23	81	24	28	81	28	1	1	1	1	1	1
	Time:	25	66.1	27.6	26.6	63.2	26.9	0.39	0.39	0.39	0.6	0.6	0.6
	Size:	793	667	769	681	691	681	17	17	17	61	61	61
Right	Iters:	-	-	-	-	-	-	-	-	-	3	7	3
	Time:	-	-	-	-	-	-	-	-	-	8.3	20.3	8.2
	Size:	-	-	-	-	-	-	-	-	-	495	756	495

Table 2. Experimental results for the Zeroconf protocol. Time in seconds.

Zeroconf protocol (Interval domain)	$K = 4$ P1	$K = 6$ P1	$K = 8$ P1	$K = 4$ P2	$K = 6$ P2	$K = 8$ P2
Time (Mass heuristic):	6.2	16.8	32.2	5.8	18.5	50.6
Time (Depth heuristic):	2.6	6.0	6.6	2.6	6.7	8.1
Time (Mix):	2.6	6.3	6.8	2.6	6.9	8.4
Time PASS:	0.6	0.8	1.1	0.7	0.9	1.2

itself may not provide good results either, with depth-unrolling and the mixed heuristics performing substantially better.

5 Conclusions

We have shown that the approach of [16,23] for abstraction of probabilistic systems can be extended to arbitrary domains, allowing probabilistic checkers to profit from well developed libraries for abstract domains like intervals, octagons, and polyhedra [3,15].

For this we have extended the construction of abstract reachability trees presented in [11] to the probabilistic case. The extension no longer yields a tree, but a stochastic 2-Player game that overapproximates the MDP semantics of the program. The correctness proof requires to use a novel technique.

The new approach allows to refine abstractions using standard techniques like delaying widenings. We have also presented a technique that selectively delays widenings using a heuristics based on quantitative properties of the abstractions.

Acknowledgements. We thank Holger Hermanns, Ernst-Moritz Hahn, and Luis M.F. Fioriti for valuable comments, Björn Wachter for many discussions during the second author's stay at the University of Oxford, made possible by Joel Ouaknine, and five anonymous reviewers for helpful remarks. The second author is supported by the DFG Graduiertenkolleg 1480 (PUMA).

References

1. PRISM homepage: http://www.prismmodelchecker.org/
2. Bagnara, R., Dobson, K., Hill, P.M., Mundell, M., Zaffanella, E.: Grids: A domain for analyzing the distribution of numerical values. In: Puebla, G. (ed.) LOPSTR 2006. LNCS, vol. 4407, pp. 219–235. Springer, Heidelberg (2007)
3. Bagnara, R., Hill, P.M., Zaffanella, E.: The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Science of Computer Programming 72(1-2), 3–21 (2008)
4. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker BLAST. Proc. of STTT 9(5-6), 505–525 (2007)
5. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In: Mogensen, T.Æ., Schmidt, D.A., Sudborough, I.H. (eds.) The Essence of Computation. LNCS, vol. 2566, pp. 85–108. Springer, Heidelberg (2002)

6. Condon, A.: The complexity of stochastic games. Inf. Comput. 96(2), 203–224 (1992)
7. Condon, A.: On algorithms for simple stochastic games. DIMACS Series in Discr. Math. and Theor. Comp. Sci., vol. 13, pp. 51–73. AMS (1993)
8. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. of POPL, pp. 238–252 (1977)
9. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL, San Antonio, Texas, pp. 269–282. ACM Press, New York (1979)
10. Esparza, J., Gaiser, A.: Probabilistic abstractions with arbitrary domains. Technical report, Technische Universität München (2011), http://arxiv.org/abs/1106.1364
11. Gulavani, B.S., Chakraborty, S., Nori, A.V., Rajamani, S.K.: Automatically refining abstract interpretations. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 443–458. Springer, Heidelberg (2008)
12. Gulavani, B.S., Rajamani, S.K.: Counterexample driven refinement for abstract interpretation. In: Hermanns, H. (ed.) TACAS 2006. LNCS, vol. 3920, pp. 474–488. Springer, Heidelberg (2006)
13. Hahn, E.M., Hermanns, H., Wachter, B., Zhang, L.: PASS: Abstraction refinement for infinite probabilistic models. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 353–357. Springer, Heidelberg (2010)
14. Hermanns, H., Wachter, B., Zhang, L.: Probabilistic CEGAR. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 162–175. Springer, Heidelberg (2008)
15. Jeannet, B., Miné, A.: APRON: A library of numerical abstract domains for static analysis. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 661–667. Springer, Heidelberg (2009)
16. Kattenbelt, M., Kwiatkowska, M.Z., Norman, G., Parker, D.: Abstraction refinement for probabilistic software. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 182–197. Springer, Heidelberg (2009)
17. Kattenbelt, M., Kwiatkowska, M.Z., Norman, G., Parker, D.: A game-based abstraction-refinement framework for markov decision processes. Form. Methods Syst. Des. 36, 246–280 (2010)
18. Monniaux, D.: Abstract interpretation of probabilistic semantics. In: SAS 2000. LNCS, vol. 1824, pp. 322–340. Springer, Heidelberg (2000)
19. Monniaux, D.: Abstract interpretation of programs as markov decision processes. In: Proc. of SAS, pp. 237–254 (2003)
20. Di Pierro, A., Hankin, C., Wiklicky, H.: On probabilistic techniques for data flow analysis. Electr. Notes Theor. Comput. Sci. 190(3), 59–77 (2007)
21. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley Interscience, Hoboken (1994)
22. Wachter, B.: Refined Probabilistic Abstraction. PhD thesis, Universität des Saarlandes (2011)
23. Wachter, B., Zhang, L.: Best probabilistic transformers. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 362–379. Springer, Heidelberg (2010)

Software Verification Using k-Induction*

Alastair F. Donaldson[1], Leopold Haller[1], Daniel Kroening[1], and Philipp Rümmer[2]

[1] Computer Science Department, Oxford University, Oxford, UK
[2] Uppsala University, Department of Information Technology, Uppsala, Sweden

Abstract. We present *combined-case k-induction*, a novel technique for verifying software programs. This technique draws on the strengths of the classical inductive-invariant method and a recent application of k-induction to program verification. In previous work, correctness of programs was established by separately proving a base case and inductive step. We present a new k-induction rule that takes an unstructured, reducible control flow graph (CFG), a natural loop occurring in the CFG, and a positive integer k, and constructs a *single* CFG in which the given loop is eliminated via an unwinding proportional to k. Recursively applying the proof rule eventually yields a loop-free CFG, which can be checked using SAT-/SMT-based techniques. We state soundness of the rule, and investigate its theoretical properties. We then present two implementations of our technique: K-INDUCTOR, a verifier for C programs built on top of the CBMC model checker, and K-BOOGIE, an extension of the Boogie tool. Our experiments, using a large set of benchmarks, demonstrate that our k-induction technique frequently allows program verification to succeed using significantly weaker loop invariants than are required with the standard inductive invariant approach.

1 Introduction

We present a novel technique for verifying imperative programs using k-induction [21]. Our method brings together two lines of existing research: the standard approach to program verification using *inductive invariants* [15], employed by practical program verifiers (including [4,5,10,20], among many others) and a recent k-induction method for program verification [12,13] which we refer to here as *split-case k-induction*. Our method, which we call *combined-case k-induction*, is directly stronger than both the inductive invariant approach and split-case k-induction. We show experimentally that combined-case k-induction frequently allows program verification to succeed using significantly weaker loop invariants than would otherwise be required, reducing annotation overhead.

We start by recapping the inductive invariant and split-case k-induction approaches to verification, and outlining our new combined-case k-induction technique. We then make the following novel contributions:

- We formally present combined-case k-induction as a proof rule operating on control flow graphs, and state soundness of the rule (§4)

* Supported by the EU FP7 STREP MOGENTES (project ID ICT-216679), the EU FP7 STREP PINCETTE (project ID ICT-257647), EPSRC projects EP/G026254/1 and EP/G051100/1, and a grant from Toyota Motors.

E. Yahav (Ed.): SAS 2011, LNCS 6887, pp. 351–368, 2011.

- We state a confluence theorem, showing that in a multi-loop program the order in which our rule is applied to loops does not affect the result of verification (§5)
- We present two implementations of our method: K-INDUCTOR, a verifier for C programs, and K-BOOGIE, an extension of the Boogie tool, and experimental results applying these tools to a large set of benchmarks (§6)

Compared with our previous work on k-induction techniques for software [12,13], which are restricted to programs containing a single *while* loop (supporting multiple loops only via a translation of all program loops to a single, monolithic loop), our novel proof rule handles multiple natural loops in *arbitrary* reducible control-flow graphs.

Throughout the paper, we are concerned with proving partial correctness with respect to assertions: establishing that whenever a statement $assert\ \ \phi$ is executed, the expression ϕ evaluates to true. We shall simply use *correctness* to refer to this notion of partial correctness.

2 Overview

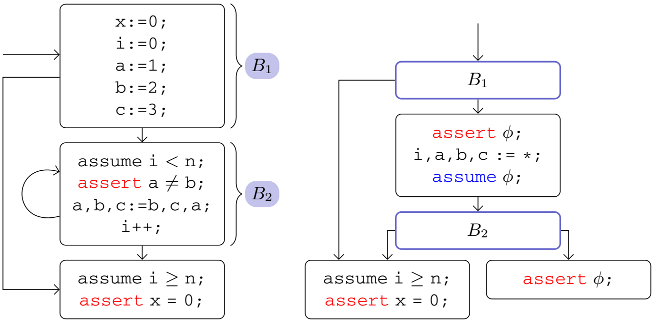
Throughout the paper, we present programs as control flow graphs (CFGs) and use the terms *program* and *CFG* synonymously. We follow the standard approach of modelling control flow using a combination of nondeterministic branches and $assume$ statements. During execution, a statement $assume\ \ \phi$ causes execution to silently (and non-erroneously) halt if the expression ϕ evaluates to false, and does nothing otherwise.

Consider the simple example program of Figure 1(a). The program initialises a, b and c to distinct values, and then repeatedly cycles their values, asserting that a and b never become equal. The condition for the loop is $i < n$, and is encoded using $assume$ statements at the start of the loop body, and at the start of the node immediately following the loop. Variable x is initialised to zero, and after the loop an assertion checks that x has not changed. The program is clearly correct.

The Inductive Invariant Approach. To formally prove a program's correctness using inductive invariants, one first associates a candidate invariant with each loop header in the program. One then shows that a) the candidate invariants are indeed loop invariants, and b) these loop invariants are strong enough to imply that no assertion in the program can fail. A technique for performing these checks in the context of unstructured programs is detailed in [3]. The technique transforms a CFG with loops into a loop-free CFG. Each loop header in the original CFG is prepended in the transformed CFG with a basic block that: asserts the loop invariant, havocs each loop-modified variable,[1] and assumes the loop invariant. Loop entry edges in the original CFG are replaced with edges to these new blocks in the transformed CFG. Each back edge in the original CFG is replaced in the transformed CFG with an edge to a new, childless basic block that asserts the invariant for the associated loop. Otherwise, the CFGs are identical.

We say that a loop is *cut* with respect to invariant ϕ. This is illustrated in Figure 1(b) for the program of Figure 1(a), where invariant ϕ is left unspecified. Cutting every loop in a CFG leads to a loop-free CFG, for which verification conditions can be computed

[1] A variable is *havocked* if it is assigned a nondeterministic value. A *loop-modified variable* is a variable that is the target of an assignment in the loop under consideration.



(a) Original CFG (b) CFG after loop cutting

Fig. 1. A simple program, and the CFG obtained using the inductive invariant approach

using weakest preconditions (an efficient method for this step is the main contribution of [3]). These verification conditions can then be discharged to a theorem prover, and if they are proven, the program is deemed correct. In Figure 1(b), taking ϕ to be $(a \neq b \wedge b \neq c \wedge c \neq a)$ allows a proof of correctness to succeed.

The main problem with the inductive invariant approach is finding the required loop invariants. Despite a wealth of research into automatic invariant generation (see [8] and references therein for a discussion of state-of-the-art techniques), this is by no means a solved problem, and in the worst case loop invariants must still be specified manually.

Split-case k-induction. The k-induction method was proposed as a technique for SAT-based verification of finite-state transition systems [21]. Let $\mathbf{I}(s)$ and $\mathbf{T}(s, s')$ be formulae encoding the initial states and transition relation for a system over sets of propositional state variables s and s', $\mathbf{P}(s)$ a formula representing states satisfying a safety property, and k a non-negative integer. To prove $\mathbf{P}$ by k-induction one must first show that $\mathbf{P}$ holds in all states reachable from an initial state within k steps, *i.e.*, that the following formula (the base case) is unsatisfiable:

$$\mathbf{I}(s_1) \wedge \mathbf{T}(s_1, s_2) \wedge \cdots \wedge \mathbf{T}(s_{k-1}, s_k) \wedge (\overline{\mathbf{P}(s_1)} \vee \cdots \vee \overline{\mathbf{P}(s_k)}) \tag{1}$$

Secondly, one must show that whenever $\mathbf{P}$ holds in k consecutive states $s_1, \ldots, s_k$, $\mathbf{P}$ also holds in the next state s_{k+1} of the system. This is established by checking that the following formula (the step case) is unsatisfiable:

$$\mathbf{P}(s_1) \wedge \mathbf{T}(s_1, s_2) \wedge \cdots \wedge \mathbf{P}(s_k) \wedge \mathbf{T}(s_k, s_{k+1}) \wedge \overline{\mathbf{P}(s_{k+1})} \tag{2}$$

In prior work [12,13] we investigated a direct lifting of k-induction from transition systems to the level of program loops. We refer to the technique of [12,13] as *split-case* k-induction, as it follows the transition system approach of splitting verification into a base case and step case. Split-case k-induction is applied to a single loop in a program. In the simplest case, no loop invariant is externally provided. Instead, assertions appearing directly in the loop body take the role of an invariant. Given a CFG containing a loop, two programs are derived; we illustrate these for our running example in Figure 2

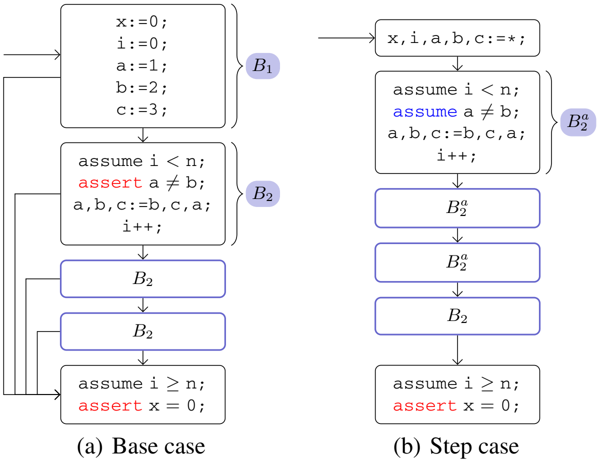


Fig. 2. Split-case k-induction, with $k = 3$

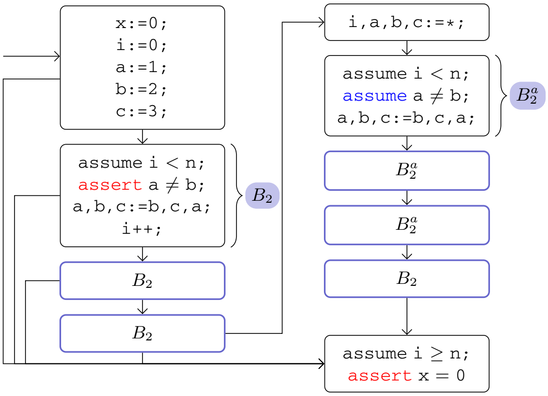


Fig. 3. Combined-case k-induction, with $k = 3$

with $k = 3$. The *base case program* (Figure 2(a)) checks that no assertion can be violated within k loop iterations. This is analogous to Equation 1 above. The *step case program* (Figure 2(b)) is analogous to Equation 2. It checks whether, after executing the loop body successfully k times from an arbitrary state, a further loop iteration can be successfully executed. In this further loop iteration, back edges to the loop header are removed, while edges that exit the loop are preserved. Thus the step case verifies that on loop exit, the rest of the program can be safely executed.

Correctness of both base and step case implies correctness of the whole program. On the other hand, an incorrect base case indicates an error; an incorrect step case might either indicate an error or a failure of k-induction to prove the program correct with the current value of k (which is, in fact, the case for the step case pictured in Figure 2(b)). In a program with multiple loops, applying split-case k-induction to one loop may lead to a base and step case that each contain loops. In this case, the splitting procedure can be applied recursively until loop-free CFGs are obtained, whose verification conditions can be discharged to a prover.

Compared with the inductive invariant approach, split-case k-induction has the advantage that verification may succeed using weaker loop invariants. The assertion $a \neq b$ in Figure 1(a) can be established using split-case k-induction as shown in Figure 2 by taking $k \geq 3$: unlike the inductive invariant approach, no external invariant (like $a \neq b \wedge b \neq c \wedge c \neq a$) is required. However, split-case k-induction has the disadvantage that in the step case (Figure 2(b)), information about the values of variables not occurring in the loop is entirely lost. Although the variable x in the example is not modified in the loop, proving the assertion $x = 0$ after the loop is beyond the reach of split-case k-induction. For split-case k-induction to succeed on this example, an invariant like $x = 0$ must be added to the loop body as an assertion. In contrast, with the inductive invariant approach, the fact that x is assigned to zero before the loop is preserved by the loop cutting process.

Our Contribution: combined-case k-induction. In *combined-case* k-induction, the strengths of split-case k-induction and the inductive invariant approach are brought together. Like the inductive invariant approach, combined-case k-induction works by cutting loops in the input CFG one at a time, resulting in a single program that needs to be checked, but like split-case k-induction, no external invariant is required.

A non-negative integer k_L is associated with each loop L in the input CFG. Loop L is then *k_L-cut* by replacing it with: k_L copies of the loop body, statements havocking all loop-modified variables, and k_L copies of the loop body where all assertions are replaced with assumptions and edges exiting the loop are removed. The last of the "assume" copies of the loop body is followed by a regular copy of the loop body, in which back edges to the loop header are removed.

Figure 3 illustrates combined-case k-induction applied to the example CFG of Figure 1(a); the single loop has been 3-cut. Comparing Figure 3 with Figure 2, observe that the base and step cases of Figure 2 are essentially merged in Figure 3. There is one key difference: variable x, which is not modified by the loop of Figure 1(a), is *not* havocked in Figure 3. Thus, unlike with split-case k-induction, we do not lose the information that the variable always retains its original value. With combined-case k-induction, the program of Figure 1(a), which is beyond the reach of split-case k-induction, can be directly verified with $k \geq 3$. As a further difference, note that base and step case are composed *sequentially,* with a transition leading from the last block B_2 of the base case to the first block of the step case. For some programs, this can increase the strength of the induction rule considerably compared to split-case k-induction, since path constraints established in the base case can be helpful for verifying the step case. Unlike with the inductive invariant approach, no external invariant is required.

Of course, combined-case k-induction does not solve the problem of finding invariants. The technique depends on invariants appearing as assertions in loop bodies. For example, if the assertion $a \neq b$ was moved to the exit of the loop in Figure 1(a), the program could not be proved using combined-case k-induction. In practice it may be necessary to strengthen the induction hypothesis by adding manually or automatically derived invariants as assertions in the body of a loop. However, our experimental evaluation in §6 demonstrates that combined-case k-induction frequently makes verification possible with significantly weaker invariants than are otherwise required, thus reducing annotation overhead.

3 Control Flow Graphs and Loops

We present our results in terms of control flow graphs, which are minimal but general enough to uniformly translate imperative programs where procedure calls are either inlined, or replaced with pre- and post-conditions. In the diagrams of §2 we presented CFGs whose nodes are basic blocks. For ease of formal presentation, from this point on we consider CFGs whose nodes are single statements.

Let X be a set of integer variables, and let Expr be the set of all integer and boolean expressions over X, using standard arithmetic and boolean operations. The set Stmt of statements over X covers nondeterministic assignments, assumptions, and assertions:

$$\mathit{Stmt} = \{x := * \mid x \in X\} \cup \{\mathit{assume}\ \phi \mid \phi \in \mathit{Expr}\} \cup \{\mathit{assert}\ \phi \mid \phi \in \mathit{Expr}\}.$$

Intuitively, a nondeterministic assignment $x := *$ alters the value of x arbitrarily; an assumption $\mathit{assume}\ \phi$ suspends program execution if ϕ is violated and can be used to encode conditional statements and constrain the effects of nondeterministic assignments, while an assertion $\mathit{assert}\ \phi$ raises an error if ϕ is violated. Neither $\mathit{assume}\ \phi$ nor $\mathit{assert}\ \phi$ have any effect if ϕ holds. We also use $x := e$ as shorthand for ordinary assignments, which can be expressed in the syntax above via a sequence of nondeterministic assignments and assumptions.

Definition 1. *A control flow graph (CFG) is a tuple $(V, \mathit{in}, E, \mathit{code})$, where V is a finite set of nodes, $\mathit{in} \in V$ an initial node, $E \subseteq V \times V$ a set of edges, and $\mathit{code} : V \to \mathit{Stmt}$ a mapping from nodes to statements.*

Loops and reducibility. We briefly recap notions of dominance, reducibility, and natural loops in CFGs, which are standard in the compilers literature [1].

Let $C = (V, \mathit{in}, E, \mathit{code})$ be a CFG. For $u, v \in V$, we say that u *dominates* v if $u = v$, or if every path from in to v must pass through u. Edge $(u, v) \in E$ is a *back edge* if v dominates u.

Definition 2. *The* natural loop *associated with back edge (u, v) is the smallest set $L_{(u,v)} \subseteq V$ satisfying the following conditions:*

- $u, v \in L_{(u,v)}$
- $(u', v') \in E \ \wedge\ v' \in L_{(u,v)} \setminus \{v\} \Rightarrow u' \in L_{(u,v)}$

For a node v such that there exists a back edge $(u, v) \in E$, the natural loop associated with v is the set $L_v = \bigcup_{u \in V, (u,v) \text{ is a back edge}} L_{(u,v)}$. Node v is the header *of loop L_v.*

For a loop $L \subseteq V$, $\mathit{modified}(L)$ denotes the set of variables that may be modified by nodes in L. Formally, $\mathit{modified}(L) = \{x \in X \mid \exists l \in L\ .\ \mathit{code}(l) = \text{‘}x := *\text{’}\}$.[2]

[2] In practice, $\mathit{modified}(L)$ could be computed more precisely, *e.g.* disregarding assignments in dead code. For a language with pointers, $\mathit{modified}(L)$ is computed with respect to an alias analysis, in the obvious way.

In a *reducible* CFG, the only edges inducing cycles are back edges. More formally, C is reducible if the CFG $C' = (V, \mathit{in}, \mathit{FE}, \mathit{code})$ is acyclic, where FE is the set $\{(u, v) \in E \mid (u, v) \text{ is not a back edge}\}$ of *forward edges*; otherwise we say that C is *irreducible*.

From now on, we assume that all CFGs are reducible. This ensures that every cycle in a CFG is part of a loop, and allows our k-induction method to work recursively, unwinding loops one-by-one until a loop-free CFG is obtained. This is not a severe restriction: structured programming techniques guarantee reducibility, and standard (though expensive) techniques exist for transforming irreducible CFGs into reducible ones [1].

Semantics. Semantically, a CFG denotes a set of execution traces, which are defined by first unwinding CFGs to prefix-closed sets of statement sequences. Subsequently, statements and statement sequences are interpreted as operations on program states.

Definition 3. *Let* $C = (V, \mathit{in}, E, \mathit{code})$ *be a CFG. The* unwinding *of* C *is defined as:*

$$\mathit{unwinding}(C) = \left\{ \begin{array}{c} \langle \mathit{code}(v_1), \ldots, \mathit{code}(v_n) \rangle \ \mid n > 0 \ \wedge \ v_1 = \mathit{in} \wedge \\ \forall i \in \{1, \ldots, n-1\}.\, (v_i, v_{i+1}) \in E \end{array} \right\} \cup \{\epsilon\} \subseteq \mathit{Stmt}^*$$

where ϵ *denotes the empty sequence.*

A *non-error state* is a store mapping variables to values in some domain $\mathcal{D}$. The set of program states for a CFG over X is the set of all stores, together with a designated error state: $S = \{\sigma \mid \sigma : X \to \mathcal{D}\} \cup \{\lightning\}$.

For an expression ϕ and store σ, we write ϕ^σ to denote the value obtained by evaluating ϕ according to the valuation of variables given by σ.

We give trace semantics to CFGs by first defining the effect of a statement on a program state. This is given by the function $\mathit{post} : S \times \mathit{Stmt} \to 2^S$ defined as follows:

$$\mathit{post}(\lightning, s) \ = \ \{\lightning\} \qquad \text{(for any statement } s\text{)}$$

For non-error states $\sigma \neq \lightning$:

$$\begin{aligned} \mathit{post}(\sigma, x := *) \ &= \ \{\sigma' \mid \sigma'(y) = \sigma(y) \text{ for all } y \neq x\} \\ \mathit{post}(\sigma, \mathit{assume}\ \phi) \ &= \ \big(\text{if } \ \phi^\sigma = \mathit{tt} \ \text{ then } \ \{\sigma\}, \ \text{ otherwise } \ \emptyset\big) \\ \mathit{post}(\sigma, \mathit{assert}\ \phi) \ &= \ \big(\text{if } \ \phi^\sigma = \mathit{tt} \ \text{ then } \ \{\sigma\}, \ \text{ otherwise } \ \{\lightning\}\big) \end{aligned}$$

The function post is lifted to the evaluation function $\mathit{traces} : S \times \mathit{Stmt}^+ \to 2^{S^*}$ on non-empty statement sequences as follows:

$$\begin{aligned} \mathit{traces}(\sigma, s) &= \{\langle \sigma, \sigma' \rangle \mid \sigma' \in \mathit{post}(\sigma, s)\} \\ \mathit{traces}(\sigma, \langle s_1, \ldots, s_n \rangle) &= \{\sigma.\tau \mid \exists \sigma'.\, \sigma' \in \mathit{post}(\sigma, s_1) \wedge \tau \in \mathit{traces}(\sigma', \langle s_2, \ldots, s_n \rangle)\} \end{aligned}$$

Here, for a state $\sigma \in S$ and state tuple $\tau \in S^m$, $\sigma.\tau \in S^{m+1}$ is the concatenation of σ and τ. The set of traces of a CFG C is the union of the traces for any of its paths:

$$\mathit{traces}(C) \ = \ \bigcup \, \{\mathit{traces}(\sigma, p) \mid \sigma \in S \setminus \{\lightning\} \wedge p \in \mathit{unwinding}(C)\}.$$

Note that there are no traces along which *assume* statements fail.

We say that CFG C is *correct* if $\lightning$ does not appear on any trace in $\mathit{traces}(C)$. Otherwise C is not correct, and a trace which leads to $\lightning$ is a counterexample to correctness.

Algorithm 1: ANALYSE

```
     Input: Reducible CFG $C = (V, in, E, code)$.
     Output: One of $\{$CORRECT, DON'T KNOW$\}$
     if $C$ is loop-free then
         if DECIDE($C$) = CORRECT then                  // Program is correct
             return CORRECT;
         else                                   // Correctness not determined
             return DON'T KNOW;
         end
     else                                          // apply the $k$-induction rule
(*)      choose loop $L$ in $C$ and depth $k \in \mathbb{N}$;
         result $\longleftarrow$ ANALYSE($C^L_k$);
         if result = CORRECT then                  // $k$-induction succeeded
             return CORRECT;
         else                                 // $k$-induction was inconclusive
(**)         either back-track to $(*)$, or return DON'T KNOW;
         end
     end
```

4 Proof Rule and Verification Algorithm

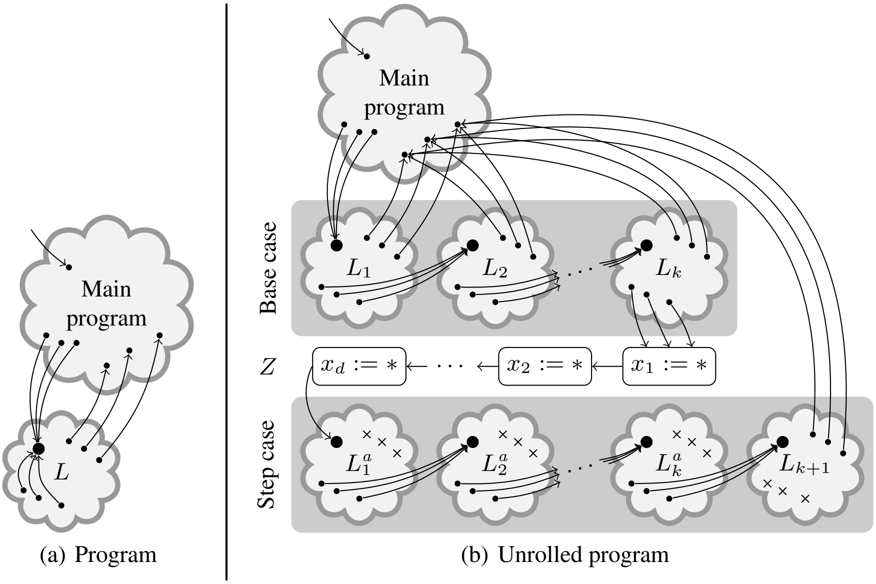
Given a CFG C containing a natural loop L (see Def. 2), and a positive integer k, we shall define a k-induction rule that transforms C into a CFG C^L_k in which loop L is eliminated via k-cutting, such that correctness of C^L_k implies correctness of C. We start by motivating the use of the rule, considering the procedure ANALYSE of Algorithm 1.

ANALYSE attempts to prove correctness of C by applying the k-induction rule recursively. At each step, a loop in the CFG, and a corresponding value of k is chosen. The loop is eliminated from the CFG by k-cutting. If the result is a loop-free CFG, correctness is checked by an appropriate decision procedure (*e.g.* an SMT solver). Otherwise, the process continues with the selection of another loop. If a k-cut CFG is not found to be correct (a recursive call to ANALYSE returns DON'T KNOW) then the procedure either returns an inconclusive result, or backtracks and applies k-induction to a different loop, and/or using a different value for k.

Note that ANALYSE cannot be used to determine that a program is incorrect. It could be modified to do so, by explicitly marking those portions of a k-cut CFG in which an error signifies a genuine bug. Genuine bugs can only be detected via traces through the k-cut CFG that do not pass through any havoc nodes introduced by the k-induction rule. Alternatively, ANALYSE can simply be executed in parallel with bounded model checking [6].

4.1 Graphical Description of k-induction Proof Rule

Figure 4(a) depicts an arbitrary CFG C that contains at least one loop, L. The CFG is separated into L (the smaller cloud), and the set of nodes outside L (the cloud labelled



(a) Program

(b) Unrolled program

Fig. 4. Schematic overview of the new k-induction rule, assuming $\mathit{modified}(L) = \{x_1, \ldots, x_d\}$

"Main program"). The main program may contain further loops, and L may contain nested loops. We assume that entry to the CFG, indicated by the edge into "Main program", is not via L. The program can be re-written to enforce this, if necessary.

Loop L has a single entry point, or *header*, indicated by the large dot in Figure 4(a). There are edges from at least one (and possibly multiple) node(s) in the main program to this header. Inside L, there are back edges from at least one node to the header. In addition, there are zero-or-more edges that exit L, leading back to the main program.

For some unspecified $k > 0$, Figure 4(b) shows the CFG C_k^L generated by our novel k-induction rule, which we present formally in §4.2. The loop L has been k-cut, producing a CFG C_k^L with four components. The nodes outside L are labelled "Main program". Edges from the main program into L in Figure 4(a) are replaced with edges into the first of k copies of the body of L, denoted $L_1, \ldots, L_k$. These are marked "Base case" in Figure 4(b). In each L_i, edges leaving L are preserved, as are edges within L, except for back edges. For $i < k$, a back edge in L is replaced in L_i with an edge to the header node of the next copy of L, namely L_{i+1}. The base case part of C_k^L checks that the first k iterations of L can be successfully executed.

In the final copy of L appearing in the base case, L_k, back edges are replaced with edges to the sequence of nodes marked Z in Figure 4(b). Z has the effect of havocking the variables $x_1, \ldots, x_d$ that comprise $\mathit{modified}(L)$, the loop-modified variables for L.

The final node of Z is followed by k copies of the body of L in which all statements of the form $\mathit{assert}\ \phi$ are replaced with $\mathit{assume}\ \phi$, and all edges leaving L are removed. These modified copies of the body of L are denoted $L_1^a, \ldots, L_k^a$ (where a denotes assume), and back-edges in L are replaced in L_i^a with edges to to the header of L_{i+1}^a, for $i < k$. In L_k^a, back edges are replaced with edges to L_{k+1}. This is a final copy of the body of L, where assertions are left intact, edges leaving L are preserved, and back-edges are removed. The fragments $L_1^a, \ldots, L_k^a$ and L_{k+1} are denoted "Step case" in Figure 4(b). Together with the Z nodes, they check that, from an arbitrary loop entry state, assuming that k iterations of L have succeeded, a further iteration that is followed by execution of the main program, will succeed.

It may be instructive to compare the abstract program of Figure 4(a), and corresponding k-cut program of Figure 4(b), with the program of Figure 1(a) and 3-cut program of Figure 3. Loop L of Figure 4(a) corresponds to B_2 in Figure 1(a). Components $L_1, \ldots, L_k$ in Figure 4(b) correspond to the three copies of B_2 on the left of Figure 3, $L_1^a, \ldots, L_k^a$ to the three copies of B_2^a on the right of Figure 3, and L_{k+1} to the additional copy of B_2 on the right of Figure 3. Finally, the Z nodes of Figure 4(b) are reflected by the statement $i, a, b, c := *$ in Figure 3.

4.2 Formal Definition of k-induction Proof Rule

We now formally define our novel k-induction rule as a transformation rule on control flow graphs, using the same notation as presented in Figure 4.

Let $C = (V, \mathit{in}, E, \mathit{code})$ be a CFG and $L \subseteq V$ a loop in C with header h. Assume that $\mathit{in} \notin L$. (This can be trivially enforced by adding an $\mathit{assume}\ \mathit{tt}$ node to C if necessary.) We present a k-induction proof rule for *positive* values of k, under the assumption that $\mathit{modified}(L)$ is non-empty. Extending the definition, and all the results presented in this paper, to allow $k = 0$, and $\mathit{modified}(L) = \emptyset$, is trivial, and the implementations we describe in §6 incorporate such extensions. However, a full presentation involves considering pedantic corner cases which make the essential concepts harder to follow without providing further insights into our work.

Thus, let $k > 0$, and suppose $\mathit{modified}(L) = \{x_1, \ldots, x_d\}$ for some $d > 0$. For $1 \leq i \leq k+1$, define $L_i = \{v_i \mid v \in L\}$. Similarly, for $1 \leq i \leq k$, define $L_i^a = \{v_i^a \mid v \in L\}$. Let $Z = \{z_1^h, \ldots, z_d^h\}$. Assume that the sets L_i $(1 \leq i \leq k+1)$, L_i^a $(1 \leq i \leq k)$ and Z consist of fresh nodes, all distinct from each other and from the nodes in V.

Definition 4. $C_k^L = (V_k^L, \mathit{in}_k^L, E_k^L, \mathit{code}_k^L)$ *is defined as follows:*

$$V_k^L = (V \setminus L) \cup \bigcup_{i=1}^{k+1} L_i \cup \bigcup_{i=1}^{k} L_i^a \cup Z$$

$$\mathit{in}_k^L = \mathit{in} \quad \textit{(recall that, by assumption, } \mathit{in} \notin L\textit{)}$$

$$E_k^L =$$

$$\begin{array}{lllr}
 & \{\, (u, v) & \mid (u, v) \in E \wedge u, v \notin L \,\} & \textit{Edges in } \text{Main program} \\
\cup & \{\, (u, h_1) & \mid (u, h) \in E \wedge u \notin L \,\} & \text{Main program} \rightarrow L_1 \\
\cup & \{\, (u_i, v_i) & \mid 1 \leq i \leq k+1 \wedge (u, v) \in E \wedge u, v \in L \wedge v \neq h \,\} & \textit{Edges in } L_i \\
\cup & \{\, (u_i^a, v_i^a) & \mid 1 \leq i \leq k \wedge (u, v) \in E \wedge u, v \in L \wedge v \neq h \,\} & \textit{Edges in } L_i^a \\
\cup & \{\, (u_i, h_{i+1}) & \mid 1 \leq i < k \wedge (u, h) \in E \wedge u \in L \,\} & L_i \rightarrow L_{i+1}\ (i < k) \\
\cup & \{\, (u_i^a, h_{i+1}^a) & \mid 1 \leq i < k \wedge (u, h) \in E \wedge u \in L \,\} & L_i^a \rightarrow L_{i+1}^a\ (i < k) \\
\cup & \{\, (u_k^a, h_{k+1}) & \mid (u, h) \in E \wedge u \in L \,\} & L_k^a \rightarrow L_{k+1} \\
\cup & \{\, (u_i, v) & \mid 1 \leq i \leq k+1 \wedge (u, v) \in E \wedge u \in L \wedge v \notin L \,\} & L_i \rightarrow \text{Main program} \\
\cup & \{\, (u_k, z_1^h) & \mid (u, h) \in E \wedge u \in L \,\} & L_k \rightarrow Z \\
\cup & \{\, (z_i^h, z_{i+1}^h) & \mid 1 \leq i < d \,\} & \textit{Edges in } Z \\
\cup & \{\, (z_d^h, h_1^a) \,\} & & Z \rightarrow L_1^a
\end{array}$$

$$\begin{aligned}
\mathit{code}^L_k(z^h_i) &= \text{‘}x_i := *\text{’} && (1 \leq i \leq d)\\
\mathit{code}^L_k(v^a_i) &= \begin{cases} \mathit{assume}\ \phi & \mathit{if}\ \mathit{code}(v) = \mathit{assert}\ \phi \\ \mathit{code}(v) & \mathit{otherwise} \end{cases} && (1 \leq i \leq k)\\
\mathit{code}^L_k(v_i) &= \mathit{code}(v) && (1 \leq i \leq k+1)\\
\mathit{code}^L_k(v) &= \mathit{code}(v) && \mathit{for}\ v \in V^L_k \cap V
\end{aligned}$$

Theorem 1 (Soundness). *If C^L_k is correct then C is correct.*

5 Theoretical Properties of the k-induction Rule

Confluence. We now turn to the question of confluence: for fixed values of k, does it matter in which order the loops of a CFG are processed when recursively applying the k-induction rule? First, we define what it means for CFGs to be isomorphic.

Definition 5. *Let $C = (V, \mathit{in}, E, \mathit{code})$ and $C' = (V', \mathit{in}', E', \mathit{code}')$ be CFGs. A bijection $\alpha : V \to V'$ is an* isomorphism *between C and C' if $\alpha(\mathit{in}) = \mathit{in}'$ and, for all $u, v \in V$, $\mathit{code}(u) = \mathit{code}'(\alpha(u))$, and $(u, v) \in E \Leftrightarrow (\alpha(u), \alpha(v)) \in E'$. If there exists an isomorphism between C and C', we say that C and C' are* isomorphic *and write $C \equiv C'$.*

It is easy to show that $\equiv$ is an equivalence relation on CFGs.

In what follows, C denotes a CFG. The next result follows directly from the definition of a natural loop:

Lemma 1. *For distinct loops L and M in C, either $L \cap M = \emptyset$, $L \subset M$ or $M \subset L$.*

Lemma 2 (Confluence of k-induction rule for disjoint loops). *Let L and M be disjoint loops in C, and let k_L and k_M be positive integers. Then $(C^L_{k_L})^M_{k_M} \equiv (C^M_{k_M})^L_{k_L}$.*

Lemma 2 shows that, for disjoint loops, the order in which k-induction is applied to each loop is irrelevant; an isomorphic CFG always results. Thus, for mutually disjoint loops $L_1, \ldots, L_d$ in a CFG C, and positive integers $k_1, \ldots, k_d$, we can write $C^{L_1,\ldots,L_d}_{k_1,\ldots,k_d}$ to denote a CFG obtained by applying the k-induction rule d times, on each application eliminating one of the loops L_i according to k_i.

Now consider loops $L \subset M$ of C, and positive integers k_L and k_M.

The CFG $C^M_{k_M}$ contains $k_M + 1$ direct copies of L, and k_M copies of L in which all assertions are replaced with assumptions. This is because L forms part of the body of M. Let us denote these copies of L by $L_1, \ldots, L_{k_M+1}$ and $L^a_1, \ldots, L^a_{k_M}$ respectively. Def. 4 ensures that they are all disjoint in $C^M_{k_M}$.

The CFG $C^L_{k_L}$ contains a loop M' identical to M, except that L has been eliminated from the body of M', and replaced with an unwinding of L proportional to k_L.

Lemma 3 (Confluence of k-induction rule for nested loops). *Let $L \subset M$ be loops of C, and k_L and k_M positive integers. Using the above notation, we have:*

$$(C^L_{k_L})^{M'}_{k_M} \equiv (C^M_{k_M})^{L_1,\ldots,L_{k+1},L^a_1,\ldots,L^a_k}_{k_L,\ldots\ldots\ldots\ldots\ldots\ldots,k_L}.$$

We now show that if we repeatedly apply the k-induction rule to obtain a loop-free CFG, as long as a value for k is used consistently for each loop in C the order in which the k-induction rule is applied to loops is irrelevant.

We assume a map *origin* which, given any CFG D derived from C by zero-or-more applications of the k-induction rule and a loop L of D, tells us the original loop in C to which L corresponds. For example, given loops $L \subset M \subset N$ in C and positive integers k_L, k_M, k_N, CFG $C^N_{k_N}$ contains many duplicates of L and M, including loops $L_1 \subset M_1$. In turn, CFG $(C^N_{k_N})^{M_1}_{k_M}$ contains many duplicates of L_1, including L_{1_1}. We have $\mathit{origin}(L_{1_1}) = \mathit{origin}(L_1) = \mathit{origin}(L) = L$, $\mathit{origin}(M_1) = \mathit{origin}(M) = M$, and $\mathit{origin}(N) = N$. Also, CFG $C^L_{k_L}$ includes loops $M' \subset N'$ identical to M and N, except that L has been unrolled. We have $\mathit{origin}(M') = M$ and $\mathit{origin}(N') = N$.

Definition 6. *Let* $\mathbf{k} : \text{(loops of } C) \rightarrow \mathbb{N}$ *associate a positive integer with each loop of* C*. For* $i \geq 0$*, let* P_i *be the set of all CFGs that can be derived from* C *by* exactly i *applications of the* k*-induction rule, together with all loop-free CFGs that can be derived from* C *by* up to i *applications of the* k*-induction rule. In all applications of the rule,* k *is chosen according to the mapping* $\mathbf{k}$.

The sequence (P_i) *is defined by* $P_0 = \{C\}$ *and*

$$\begin{aligned} P_i = \; & \{D^L_{\mathbf{k}(\mathit{origin}(L))} \mid D \in P_{i-1} \wedge L \textit{ is a loop of } D\} \cup \qquad (\textit{for } i > 0) \\ & \{D \in P_{i-1} \qquad \mid D \textit{ is loop free } \}\,. \end{aligned}$$

Our main confluence theorem states that the result of exhaustively applying the combined-case k-induction rule is independent (up to isomorphism) of the order in which loops are eliminated. The result is stated with respect to Def. 6, and is proved using Lemmas 1–3.

Theorem 2 (Global confluence). *There is an integer* n *such that* $P_m = P_n$ *for all* $m \geq n$*. All the CFGs in* P_n *are isomorphic, and loop-free.*

It should be noted that, although the final CFGs are isomorphic regardless of the order of loop elimination, intermediate CFGs can differ both in size and in the number of remaining loops. Also, the total number of required applications of the k-induction rule depends on this order: eliminating loops starting from innermost loops will altogether need fewer rule applications than elimination starting with outer loops.

Size of Loop-free Programs Produced by k-induction. Since the program C^L_k obtained via a single application of the k-induction rule contains $2k+1$ copies of the loop L, repeated application can increase the size of a program exponentially. Such exponential growth can only occur in the presence of nested loops, however, because k-induction leaves program parts outside of the eliminated loop L unchanged. By a simple complexity analysis, we find that the size of loop-free programs derived though repeated application of k-induction is (singly) exponential in the depth of the deepest loop nest in the worst case, but only linear in the number of disjoint loops. Thus the size of generated programs is not a bottleneck for combined-case k-induction in practice.

6 Experimental Evaluation

We have implemented our techniques in two tools. K-BOOGIE is an extension of the BOOGIE verifier, allowing programs written in the BOOGIE language to be verified using combined-case k-induction, as well as with the inductive invariant approach supported by regular BOOGIE. When combined-case k-induction is selected, our novel k-induction rule is used to eliminate innermost loops first. As BOOGIE is an intermediate language for verification, K-BOOGIE can be applied to programs originating from several different languages, including Spec# [4], Dafny [20], Chalice, VCC, and Havoc. K-INDUCTOR is a k-induction-based verifier for C programs built on top of the CBMC tool [9]. K-INDUCTOR supports both split- and combined-case k-induction. Again, with combined-case k-induction, loops are processed innermost first. With split-case k-induction, all outermost loops are simultaneously eliminated in each application of the k-induction rule; we have found this strategy works best in practice.

We use K-BOOGIE to compare the standard inductive invariant approach to verification with our novel combined-case k-induction method, and K-INDUCTOR to compare combined-case k-induction with split-case k-induction. Both tools, and all our benchmarks, are available online: **http://www.cprover.org/kinduction**

Experiments with K-BOOGIE. We apply K-BOOGIE to a set of 26 Boogie programs, the majority of which were machine-generated from (hand-written) Dafny programs included in the Boogie distribution. Most of the programs verify functional correctness of standard algorithms, including sophisticated procedures such as the Schorr-Waite graph marking algorithm. The Boogie programs contain altogether 40 procedures, annotated with assertions, pre-/post-conditions, and loop invariants, and were not previously known to be amenable to k-induction. Six of the procedures contain multiple loops, three contain (singly) nested loops. Our findings are summarised in Table 1.

To evaluate the applicability of k-induction, we first split conjunctive loop invariants in the programs into multiple invariants, and then eliminated all invariants that were not necessary to verify assertions and post-conditions even with the normal Boogie induction rule. Since Boogie uses abstract interpretation (primarily with an interval domain) to automatically infer simple invariants, in this step also all those invariants were removed that can be derived with the help of inexpensive abstract interpretation techniques. The elimination of invariants was done in a greedy manner, so that in the end a minimum set of required invariants for each procedure was obtained (though not necessarily a set with the smallest number of invariants).

We then checked, using $0 \leq k \leq 4$, which of the loop invariants were unnecessary with combined-case k-induction. This was done by first trying to remove invariants individually, keeping all other invariants of a procedure. In Table 1, **# removable** shows the number of invariants that could be individually removed, in comparison to the total number of invariants. As second step, we determined maximum sets of invariants that could be removed simultaneously, shown under **# sim. remov.** in Table 1. In both cases, we show the largest value of k required for invariant removal, over all loops (**required** k), which was determined by systematically enumerating all combinations of k. For each procedure, we show the verification time with the normal Boogie loop rule (**time w/o k-ind.**), the range of times needed by the various runs with k-induction (**times w/**

Table 1. Experimental results applying K-BOOGIE to Dafny and Boogie benchmarks included in the Boogie distribution

Procedure	# removable,	# sim. remov.,	time w/o	times w/	LOC/	LOC
	required k	required k	k-ind.	k-ind.	# loops	program
Procedures generated from Dafny programs						
VSI-b1.Add	2/4, 1	2/4, 1	2.5s	[2.6s, 2.9s]	114/2	710
VSI-b2.BinarySearch	0/5, 1		2.5s	2.6s	100/1	595
VSI-b3.Sort	1/16, 1	1/16, 1	3.9s	[6.2s, 6.2s]	186/2	798
VSI-b3.RemoveMin	1/6, 1	1/6, 1	3.0s	[4.5s, 4.5s]	176/2	
VSI-b4.Map.FindIndex	3/4, 2	2/4, 1	3.7s	[3.6s, 4.6s]	84/1	956
VSI-b6.Client.Main	1/3, 1	1/3, 1	3.1s	[3.5s, 3.5s]	139/1	900
VSI-b8.Glossary.Main	4/16, 1	3/16, 1	5.3s	[18.7s, 21.6s]	381/3	1998
VSI-b8.Glossary.readDef	0/1, 1		3.4s	3.6s	71/1	
VSI-b8.Map.FindIndex	0/1, 1		3.3s	3.4s	66/1	
Composite.Adjust	1/3, 2	1/3, 2	5.3s	[44.3s, 44.3s]	80/1	1275
LazyInitArray	1/5, 1	1/5, 1	5.0s	5.0s	165/1	806
SchorrWaite.RecursiveMark	0/6, 1		3.4s	4.2s	98/1	1175
SchorrWaite.IterativeMark	2/17, 1	2/17, 1	4.8s	5.7s	177/1	
SchorrWaite.Main	4/27, 1	3/27, 1	33.3s	[16.9s, 34.5s]	275/1	
SumOfCubes.Lemma0	1/2, 1	1/2, 1	2.6s	[2.6s, 2.6s]	81/1	915
SumOfCubes.Lemma1	1/2, 1	1/2, 1	2.7s	[2.7s, 2.7s]	65/1	
SumOfCubes.Lemma2	1/2, 1	1/2, 1	2.5s	[2.5s, 2.5s]	48/1	
SumOfCubes.Lemma3	1/2, 1	1/2, 1	2.5s	[2.5s, 2.5s]	51/1	
Substitution	0/1, 1		2.7s	2.8s	131/1	846
PriorityQueue.SiftUp	1/2, 2	1/2, 2	2.9s	3.3s	92/1	819
PriorityQueue.SiftDown	1/2, 2	1/2, 2	3.1s	[16.0s, 16.0s]	101/1	
MatrixFun.MirrorImage	2/6, 1	2/6, 1	2.9s	[3.5s, 3.5s]	125/2	922
MatrixFun.Flip	1/3, 1	1/3, 1	2.7s	[2.8s, 2.8s]	103/1	
ListReverse	2/3, 2	2/3, 2	2.4s	[2.4s, 2.4s]	71/1	329
ListCopy	1/4, 1	1/4, 1	2.5s	[2.5s, 2.5s]	141/1	434
ListContents	1/3, 1	1/3, 1	3.4s	[6.1s, 6.1s]	141/1	717
Cubes	3/4, 2	3/4, 4	2.8s	[2.7s, 3.3s]	97/1	339
Celebrity.FindCelebrity1	1/1, 2	1/1, 2	2.5s	[3.0s, 3.0s]	98/1	795
Celebrity.FindCelebrity2	0/1, 1		2.5s	2.7s	99/1	
Celebrity.FindCelebrity3	0/2, 1		2.5s	2.6s	86/1	
VSC-SumMax	1/2, 1	1/2, 1	2.4s	[2.7s, 2.7s]	77/1	458
VSC-Invert	0/1, 1		15.4s	[3.2s, 3.2s]	61/1	568
VSC-FindZero	1/2, 1	1/2, 1	2.7s	[2.7s, 2.7s]	90/1	625
VSC-Queens.CConsistent	0/3, 1		2.6s	2.7s	79/1	825
VSC-Queens.SearchAux	0/1, 1		2.9s	3.2s	139/1	
Native Boogie programs						
StructuredLocking	1/1, 1	1/1, 1	1.8s	[1.8s, 1.8s]	16/1	40
StructuredLockingWithCalls	0/1, 1		1.8s	1.8s	13/1	
Structured.RunOffEnd1	1/1, 1	1/1, 1	1.8s	[1.8s, 1.8s]	12/1	53
BubbleSort	7/14, 1	7/14, 1	2.1s	[3.0s, 3.2s]	33/3	42
DutchFlag	1/5, 1	1/5, 1	1.9s	[2.0s, 2.0s]	29/1	37

Table 2. Experimental results applying K-INDUCTOR to DMA processing benchmarks

Benchmark	LOC	# loops	nesting	split-case				combined-case			speedup with
			depth	min k	max k	time (s)	# invariants	min k	max k	time (s)	combined-case
1-buf	151	2	2	1	1	0.32	3	0	1	0.23	1.35
1-buf I/O	178	2	2	1	1	0.39	5	0	1	0.27	1.44
2-buf	254	3	2	1	2	1.18	17	0	2	0.45	2.62
2-buf I/O	304	3	2	1	2	2.06	29	0	2	0.53	3.85
3-buf	282	4	2	1	3	9.56	27	0	3	1.00	9.58
3-buf I/O	364	4	2	1	3	8.47	38	0	3	1.03	8.19
Euler simple	101	3	3	1	2	3.30	10	0	2	2.95	1.12
sync atomic op	91	3	2	1	1	0.40	4	0	1	0.18	2.24
sync mutex	83	2	2	1	1	0.87	2	0	1	0.28	3.08

k**-ind.**, using the smallest k for which verification succeeded), the number of lines of executable code (**LOC**), and the number of loops (**# loops**). We also show the total number of lines for each program (**LOC program**), including all procedures and additional definitions (which can be quite considerable). Experiments were run on a 2.5GHz Intel Core2 Duo machine with 2 GB RAM and Windows Vista.

For all but 11 of the procedures, spread over 22 of the 26 programs, we find that, with 1- or 2-induction, we are able to remove invariants that are necessary for the normal Boogie loop rule. Since the normal Boogie rule corresponds to 0-induction, augmented with assumptions and assertions encoding the loop invariant, already 1-induction is often able to succeed with a significantly reduced number of invariants. Values of k larger than two proved to be beneficial only for a single procedure (Cubes); additional gains with even larger values of k therefore seem unlikely. On average, 32% of the invariants could be removed (simultaneously) when using k-induction. The verification times with k-induction are only marginally larger than those with the normal Boogie rule, and for some examples even smaller (averaging over all benchmarks, verification time increased by 44%). In cases where the same set of invariants is used, verification times almost coincide. This shows that k-induction, with small values of k, can be useful for general-purpose verification, since the (extremely time-consuming) process of constructing inductive invariants can be shortened.

Experiments with K-INDUCTOR. We apply K-INDUCTOR to a set of benchmarks from the domain of direct memory access (DMA) race checking, studied in [12,13] (in which full details can be found). These consist of data processing programs for the Cell BE processor, where data is manipulated using DMA. In [12,13], split-case k-induction is applied to these benchmarks, under the simplifying assumption that in many cases inner loops unrelated to DMA are manually sliced away, leaving single-loop programs. We find that combined-case k-induction allows us to handle inner loops in these benchmarks directly. With split-case k-induction, handling inner loops requires the addition of numerous invariants, as assertions in the program text.

For each DMA processing benchmark, Table 2 shows the number of lines of code (**LOC**), the number of loops processed by k-inductor (**# loops**, this is the number of loops after function inlining, which may cause loop duplication), and the depth of the deepest loop nest (**nesting depth**). All benchmarks involve nested loops. For the split-case and combined-case approaches, we manually determined the smallest values of k required for each loop in order for verification to succeed. In each case, the minimum and maximum values of k required are shown (**min/max** k), as well as the time (in seconds) taken for verification with this combination of k values (**time**). For the split-case approach, we show the number of invariants that had to be added manually for verification to succeed (**# invariants**) – these invariants are *not* required when our novel combined-case method is employed. Finally, we show the speedup obtained by using combined-case k-induction instead of split-case k-induction (**speedup with combined-case**). Experiments are performed on a 3GHz Intel Xeon machine, 40 GB RAM, 64-bit Linux. MiniSat 2 is used as a back-end SAT solver for CBMC. Manually specified invariants are mainly simple facts related to variable ranges; many could be inferred automatically using abstract interpretation.

The results show that combined-case k-induction avoids the need for a significant number of additional invariants when verifying these examples. This allows many inner loops that are unrelated to DMA processing (and thus do not contain assertions of interest) to be handled using $k = 0$. In such cases, $k = 0$ is sufficient because we do not havoc variables that are not modified by the loop in question. With split-case k-induction, explicit invariant assertions must be added to assert that such variables are invariant under loop execution. This involves a lot of manual effort, and verification with k-induction requires at least $k = 1$ to take advantage of these assertions.

We also find that combined-case k-induction is uniformly, and sometimes significantly faster than split-case k-induction. We attribute this to the multiple loop-free programs that must be solved with split-case k-induction, compared with the single loop-free program associated with combined-case k-induction. Verification using combined-case k-induction never takes longer than three seconds for this benchmark set, suggesting good scalability of the approach.

7 Related Work and Conclusions

The concept of k-induction was first published in [21,7], targeting the verification of hardware designs and transition relations. A major emphasis of these two papers is on the restriction to loop-free or shortest paths, which is so far not considered in our k-induction rule due to the size of state vectors and the high degree of determinism in software. Several optimisations and extensions to the technique have been proposed, including property strengthening to reduce induction depth [22], improving performance via incremental SAT solving [14], and verification of temporal properties [2].

Besides hardware verification, k-induction has been used to analyse synchronous programs [18,16] and, recently, SystemC designs [17]. To the best of our knowledge, the first application of k-induction to imperative software programs was done in the context of DMA race checking [12,13], from which we also draw some of the benchmarks used in this paper. A combination of the k-induction rule of [12,13], abstract interpretation, and domain-specific invariant strengthening techniques for DMA race analysis is the topic of [11]. The main new contributions of this paper over our previous work are that we present a k-induction technique that can be applied in a structured manner to multiple, *arbitrary* loops in a reducible control-flow graph (prior work was restricted to single-loop programs, with multiple loops handled via translation to a single, monolithic loop), and we present the novel idea of combined-case k-induction, where base and step case are combined into a single program. We have demonstrated experimentally that combined-case k-induction can allow verification to succeed using weaker loop invariants than are required with either split-case k-induction or the inductive invariant approach, and that it can significantly out-perform split-case k-induction.

Combined-case k-induction depends on analysis of those variables which are not modified by a given loop. This can be viewed as a simple kind of loop summary. We plan to investigate whether k-induction can be strengthened using more sophisticated loop summarisation analyses [19]. In addition, we intend to study automatic techniques for selecting and exploring values of k for programs with multiple loops.

Finally, our experiments show that the effectiveness of k-induction varies significantly from example to example. It would be interesting and useful to characterise, ideally formally but at least intuitively, classes of programs for which k-induction is likely to be beneficial for verification. Our initial efforts in this direction indicate that it is a challenging problem.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison Wesley, Reading (2006)
2. Armoni, R., Fix, L., Fraer, R., Huddleston, S., Piterman, N., Vardi, M.Y.: SAT-based induction for temporal safety properties. Electr. Notes Theor. Comput. Sci. 119(2), 3–16 (2005)
3. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In: PASTE, pp. 82–87. ACM, New York (2005)
4. Barnett, M., Leino, K.R.M., Schulte, W.: The spec# programming system: An overview. In: Barthe, G., et al. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
5. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
6. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers 58, 118–149 (2003)
7. Bjesse, P., Claessen, K.: SAT-based verification without state space traversal. In: Johnson, S.D., Hunt Jr., W.A. (eds.) FMCAD 2000. LNCS, vol. 1954, pp. 372–389. Springer, Heidelberg (2000)
8. Bradley, A.R., Manna, Z.: Property-directed incremental invariant generation. Formal Asp. Comput. 20(4-5), 379–405 (2008)
9. Clarke, E., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
10. Distefano, D., Parkinson, M.J.: jStar: towards practical verification for Java. In: OOPSLA, pp. 213–226. ACM, New York (2008)
11. Donaldson, A.F., Haller, L., Kroening, D.: Strengthening Induction-Based Race Checking with Lightweight Static Analysis. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 169–183. Springer, Heidelberg (2011)
12. Donaldson, A.F., Kroening, D., Rümmer, P.: Automatic analysis of scratch-pad memory code for heterogeneous multicore processors. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 280–295. Springer, Heidelberg (2010)
13. Donaldson, A.F., Kroening, D., Rümmer, P.: Automatic analysis of DMA races using model checking and k-induction. In: Formal Methods in System Design (to appear, 2011), doi: 10.1007/s10703-011-0124-2
14. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electr. Notes Theor. Comput. Sci. 89(4) (2003)
15. Floyd, R.: Assigning meaning to programs. In: Mathematical Aspects of Computer Science. Proceedings of Symposia in Applied Mathematics, vol. 19, pp. 19–32. AMS, Providence (1967)
16. Franzén, A.: Using satisfiability modulo theories for inductive verification of Lustre programs. Electr. Notes Theor. Comput. Sci. 144(1), 19–33 (2006)
17. Große, D., Le, H.M., Drechsler, R.: Proving transaction and system-level properties of untimed SystemC TLM designs. In: MEMOCODE, pp. 113–122. IEEE, Los Alamitos (2010)

18. Hagen, G., Tinelli, C.: Scaling up the formal verification of Lustre programs with SMT-based techniques. In: FMCAD, pp. 109–117. IEEE, Los Alamitos (2008)
19. Kroening, D., Sharygina, N., Tonetta, S., Tsitovich, A., Wintersteiger, C.M.: Loop summarization using abstract transformers. In: Cha, S(S.), Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 111–125. Springer, Heidelberg (2008)
20. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 348–370. Springer, Heidelberg (2010)
21. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Johnson, S.D., Hunt Jr., W.A. (eds.) FMCAD 2000. LNCS, vol. 1954, pp. 108–125. Springer, Heidelberg (2000)
22. Vimjam, V.C., Hsiao, M.S.: Explicit safety property strengthening in SAT-based induction. In: VLSID, pp. 63–68. IEEE, Los Alamitos (2007)

Using Bounded Model Checking to Focus Fixpoint Iterations[★]

David Monniaux[1] and Laure Gonnord[2]

[1] CNRS, VERIMAG, Gières, France
[2] Université Lille 1, LIFL, Villeneuve d'Ascq, France

Abstract. Two classical sources of imprecision in static analysis by abstract interpretation are widening and merge operations. Merge operations can be done away by distinguishing paths, as in trace partitioning, at the expense of enumerating an exponential number of paths.

In this article, we describe how to avoid such systematic exploration by focusing on a single path at a time, designated by SMT-solving. Our method combines well with acceleration techniques, thus doing away with widenings as well in some cases. We illustrate it over the well-known domain of convex polyhedra.

1 Introduction

Program analysis aims at automatically checking that programs fit their specifications, explicit or not — e.g. "the program does not crash" is implicit. Program analysis is impossible unless at least one of the following holds: it is unsound (some violations of the specification are not detected), incomplete (some correct programs are rejected because spurious violations are detected), or the state space is finite (and not too large, so as to be enumerated explicitly or implicitly). *Abstract interpretation* is sound, but incomplete: it over-approximates the set of behaviours of the analysed program; if the over-approximated set contains incorrect behaviours that do not exist in the concrete program, then false alarms are produced. A central question in abstract interpretation is to reduce the number of false alarms, while keeping memory and time costs reasonable [8].

Our contribution is a method leveraging the improvements in SMT-solving to increase the precision of invariant generation by abstract fixpoint iterations. On practical examples from the literature and industry, it performs better than previous generic technique and is less "ad-hoc" than syntactic heuristics found in some pragmatic analyzers.

The first source of imprecision in abstract interpretation is the choice of the set of properties represented inside the analyser (the *abstract domain*). Obviously, if the property to be proved cannot be reflected in the abstract domain (e.g. we wish to prove a numerical relation but our abstract domain only considers Boolean variables), then the analysis cannot prove it.

In order to prove that there cannot be a division by zero in the first branch of the second if-then-else of Listing 1, one would need the non-convex property that $x \geq 0.01 \vee x \leq -0.01$. An analysis representing the invariant at that point in a domain of

[★] This research was partially funded by ANR project "ASOPT".

E. Yahav (Ed.): SAS 2011, LNCS 6887, pp. 369–385, 2011.

Listing 1. C implementation of $y = \sin(x)/x - 1$, with the $-0.01 \leq x \leq 0.01$ range implemented using a Taylor expansion around zero in order to avoid loss of precision and division by zero as $\sin(x) \simeq x \to 0$.

```
if (x >= 0) { xabs = x; } else { xabs = -x; }
if (xabs >= 0.01) {
  y = sin(x) / x - 1;
} else {
  xsq = x*x;  y = xsq*(-1/6. + xsq/120.);
}
```

Listing 2. Circular buffer indexing

```
int x = 0;
while (true) {
  if (nondet()) {
    x = x+1;
    if (x >= 100) x = 0;
} }
```

convex properties (intervals, polyhedra, etc.) will fail to prove the absence of division by zero (incompleteness).

Obviously, we could represent such properties using disjunctions of convex polyhedra, but this leads to combinatorial explosion as the number of polyhedra grows: at some point heuristics are needed for merging polyhedra in order to limit their number; it is also unclear how to obtain good widening operators on such domains. The same expressive power can alternatively be obtained by considering all program paths separately ("merge over all paths") and analysing them independently of each other. In order to avoid combinatorial explosion, the *trace partitioning* approach [36] applies merging heuristics. In contrast, our method relies on the power of modern SMT-solving techniques.

The second source of imprecision is the use of *widening operators* [14]. When analysing loops, static analysis by abstract interpretation attempts to obtain an *inductive invariant* by computing an increasing sequence $X_1, X_2, \ldots$ of sets of states, which are supersets of the sets of states reachable in at most $1, 2, \ldots$ iterations. In order to enforce convergence within finite time, the most common method is to use a widening operator, which extrapolates the first iterates of the sequence to a candidate limit. Optional narrowing iterations may regain some precision lost by widening.

Illustrating Example. Consider Listing 2, a simplification of a fragment of an actual industrial reactive program: indexing of a circular buffer used only at certain iterations of the main loop of the program, chosen non-deterministically. If the non-deterministic choice nondet() is replaced by **true**, analysis with widening and narrowing finds $[0, 99]$. Unfortunately, the "narrowing" trick is brittle, and on Listing 2, widening yields $[0, +\infty)$, and this is not improved by narrowing![1] In contrast, our semantically-based method would compute the $[0, 99]$ invariant on this example by first *focusing* on the following path inside the loop:

Listing 3. Example focus path

```
assume(nondet());   x = x+1;   assume(x < 100);
```

If we wrap this path inside a loop, then the least inductive invariant is $[0, 99]$. We then check that this invariant is inductive for the original loop.

This is the basic idea of our method: it performs fixpoint iterations by focusing temporarily on certain paths in the program. In order to obtain the next path, it performs bounded model checking using SMT-solving.

2 Background and Notations in Abstract Interpretation

We consider programs defined by a control flow graph: a set P of control points, for each control point $p \in P$ a (possibly empty) set I_p of initial values, a set $E \subseteq P \times P$ of directed edges, and the semantics $\tau_e : \mathcal{P}(\Sigma) \rightarrow \mathcal{P}(\Sigma)$ of each edge $e \in E$ where $\mathcal{P}(\Sigma)$ is the set of possible values of the tuple of program variables. τ_e thus maps a set of states before the transition expressed by edge e to the set of states after the transition.

To each control point $p \in P$ we attach a set $X_p \subseteq \Sigma$ of reachable values of the tuple of program variables at program point p. The concrete semantics of the program is the least solution of a system of semantic equations [14]: $X_p = I_p \cup \bigcup_{(p',p) \in E} \tau_{(p',p)}(X_{p'})$.

Abstract interpretation replaces the concrete sets of states in $\mathcal{P}(\Sigma)$ by elements of an abstract domain D. In lieu of applying exact operations τ to sets of concrete program states, we apply abstract counterparts $\tau^\sharp$.[2] An abstraction $\tau^\sharp$ of a concrete operation τ is deemed to be correct if it never "forgets" states:

$$\forall X \in D \; \tau(X) \subseteq \tau^\sharp(X) \tag{1}$$

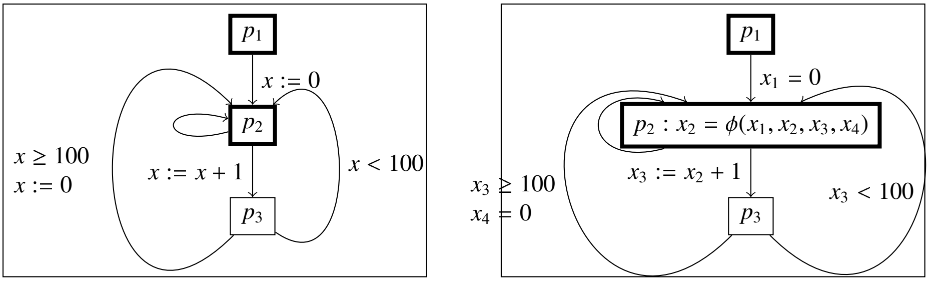
We also assume an "abstract union" operation $\sqcup$, such that $X \cup Y \subseteq X \sqcup Y$. For instance, Σ can be $\mathbb{Q}^n$, D can be the set of convex polyhedra and $\sqcup$ the convex hull operation [27,17,3].

In order to find an inductive invariant, one solves a system of abstract semantic inequalities:

$$\begin{cases} \forall p \; I_p \subseteq X_p \\ \forall (p', p) \in E \; \tau^\sharp_{(p',p)}(X_{p'}) \subseteq X_p. \end{cases} \tag{2}$$

[1] On this example, it is possible to compute the $[0, 99]$ invariant by so called "widening upto" [28, Sec. 3.2], or with "thresholds" [8]: essentially, the analyser notices syntactically the comparison $x < 100$ and concludes that 99 is a "good value" for x, so instead of widening directly to $+\infty$, it first tries 99. This method only works if the interesting value is a syntactic constant.

[2] Many presentations of abstract interpretation distinguish the abstract element $x^\sharp \in D$ from the set of states $\gamma(x^\sharp)$ it represents. We opted not to, for the sake of brevity.



(a) With original variables

(b) SSA version. $x = \phi(e_1, e_2, \ldots)$ denotes a SSA ϕ-node: x takes value e_1 if control flows from the first incoming edge, e_2 from the second...

Fig. 1. Control flow graph corresponding to listing 2

Since the $\tau_e^\sharp$ are correct abstractions, it follows that any solution of such a system defines an inductive invariant; one wishes to obtain one that is as strong as possible ("strong" meaning "small with respect to $\sqsubseteq$"), or at least sufficiently strong as to imply the desired properties.

Assuming that all functions $\tau_e^\sharp$ are monotonic with respect to $\sqsubseteq$, and that $\sqcup$ is the least upper bound operation in D with respect to $\sqsubseteq$, one obtains a system of monotonic abstract equations: $X_p = I_p \sqcup \bigsqcup_{(p',p) \in E} \tau^\sharp_{(p',p)}(X_{p'})$. If $(D, \sqsubseteq)$ has no infinite ascending sequences ($d_1 \subsetneq d_2 \subsetneq \ldots$ with $d_1, d_2, \cdots \in D$), then one can solve such a system by iteratively replacing the contents of the variable on the left hand side by the value of the right hand side, until a fixed point is reached. The order in which equations are iterated does not change the final result.

Many interesting abstract domains, including that of convex polyhedra, have infinite ascending sequences. One then classically uses an extrapolation operator known as *widening* and denoted by $\triangledown$ in order to enforce convergence within finite time. The iterations then follow the "upward iteration scheme":

$$X_p := X_p \triangledown \left(X_p \sqcup \bigsqcup_{(p',p) \in E} \tau^\sharp_{(p',p)}(X_{p'}) \right) \tag{3}$$

where the contents of the left hand side gets replaced by the value of the right hand side. The convergence property is that any sequence u_n of elements of D of the form $u_{n+1} = u_n \triangledown v_n$, where v_n is another sequence, is stationary [14]. It is sufficient to apply widening only at a set of program control nodes P_W such that all cycles in the control flow graph are cut. Then, through a process of *chaotic iterations* [13, Def. 4.1.2.0.5, p. 127], one converges within finite time to an inductive invariant satisfying Rel. 2.

Once an inductive invariant is found, it is possible to improve it by iterating the $\psi^\sharp$ function defined as $Y = \psi^\sharp(X)$, noting $X = (X_p)_{p \in P}$ and $Y = (Y_p)_{p \in P}$, with $Y_p = I_p \sqcup \bigsqcup_{(p',p) \in E} \tau^\sharp_{(p',p)}(X_{p'})$. If X is an inductive invariant, then for any k, ${\psi^\sharp}^k(X)$ is also an invariant. This technique is an instance of *narrowing iterations*, which may help recover some of the imprecision induced by widening [14, §4].

Algorithm 1. Classical Algorithm

```
$A \leftarrow \emptyset$;
for all $p \in P$ such that $I_p \neq \emptyset$ do
    $A \leftarrow A \cup \{p\}$
end for;                                   ▷ Initialise $A$ to the set of all non empty initial nodes
while $A$ is not empty do                  ▷ Fixpoint Iteration
    Choose $p_1 \in A$
    $A \leftarrow A \setminus \{p_1\}$
    for all outgoing edge (e) from $p_1$ do
        Let $p_2$ be the destination of $e$ :
        if $p_2 \in P_W$ then
            $X_{temp} \leftarrow X_{p_2} \nabla (X_{p_2} \sqcup \tau_e^\sharp(X_{p_1}))$     ▷ Widening node;
        else
            $X_{temp} \leftarrow X_{p_2} \sqcup \tau_e^\sharp(X_{p_1})$ ;
        end if
        if $X_{temp} \quad X_{p_2}$ then          ▷ The value must be updated
            $X_{p_2} \leftarrow X_{temp}$;
            $A \leftarrow A \cup \{p_2\}$;
        end if
    end for;
end while;                                 ▷ End of Iteration
Possibly narrow
return all $X_{p_i}$s;
```

A naive implementation of the upward iteration scheme described above is to maintain a work-list of program points p such that X_p has recently been updated and replaced by a strictly larger value (with respect to $\subseteq$), pick and remove the foremost member p, apply the corresponding rule $X_p := \ldots$, and insert into the work-list all p' such that $(p, p') \in E$ (This algorithm is formally described in Algorithm 1).

Example of Section 1 (Cont'd) Figure 1(a) gives the control flow graph obtained by compilation of Listing 2. Node p_2 is the unique widening node.

The classical algorithm (with the interval abstract domain) performs on this control flow graph of the following iterations :

- Initialisation : $X_{p_1} \leftarrow (-\infty, +\infty), X_{p_2} \leftarrow X_{p_3} \leftarrow X_{p_4} \leftarrow \emptyset$.
- Step 1: $X_{p_2} \leftarrow [0, 0]$, then the transition to p_3 is enabled, $X_{p_3} \leftarrow [1, 1]$, then the return edge to p_2 gives the new point $x = 1$ to X_{p_2}, the new polyhedron is then $X_{p_2} = [0, 1]$ after performing the convex hull. Widening gives the polyhedron $X_{p_2} = [0, \infty)$.
  (The widening operator on intervals is defined as $[x_l, x_r] \nabla [x'_l, x'_r] = [x"_l, x"_r]$ where $x"_l = x_l$ if $x_l = x'_l$ else $-\infty$, and $x"_r = x_r$ if $x_r = x'_r$ else $+\infty$.)
- Step 2: X_{p_3} becomes $[1, +\infty)$. The second transition from p_3 to p_2 is thus enabled, and the back edge to p_2 gives the point $x = 0$ to X_{p_2}. At the end of step 2 the convergence is reached.
- If we perform a narrowing sequence, there is no gain of precision because of the simple loop over the control point p_2.

3 Our Method

We have seen two examples of programs where classical polyhedral analysis fails to compute good invariants. How could we improve on these results?

- In order to get rid of the imprecision in Listing 1, one could "explode" the control-flow graph: in lieu of a sequence of n if-then-else, with n merge nodes with 2 input edges, one could distinguish the 2^n program paths, and having a single merge node with 2^n input edges. As already pointed out, this would lead to exponential blowup in both time and space.
- One way to get rid of imprecision of classical analysis (Sec. 2) on the program from Fig. 1(a) would be to consider each path through the loop at a time and compute a local invariant for this path. Again, the number of such paths could be exponential in the number of tests inside the loop.

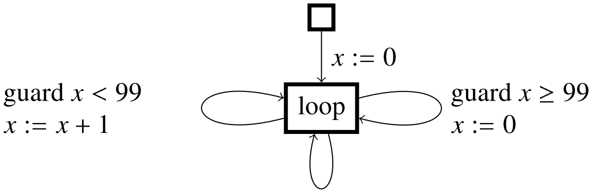
The contribution of our article is a generic method that addresses both of these difficulties.

3.1 Reduced Transition Multigraph and Path Focusing

Consider a control flow graph (P, E) with associated transitions $(\tau_e)_{e \in E}$, a set of *widening points* $P_W \subseteq P$ such that removing P_W cuts all cycles in the graph, and a set P_R of *abstraction points*, such that $P_W \subseteq P_R \subseteq P$ (On the figures, the nodes in P_R are in bold). We make no assumption regarding the choice of P_W; there are classical methods for choosing widening points [9, §3.6]. P_R can be taken equal to P_W, or may include other nodes; this makes sense only if these nodes have several incoming edges. Including other nodes will tend to reduce precision, but may improve scalability. We also make the simplifying assumption that the set of initial values I_p is empty for all nodes in $P \setminus P_R$ — in other words, the set of possible control points at program start-up is included in P_R.

We construct (virtually) the reduced control multigraph (P_R, E_R), with edges E_R consisting of the paths in (P, E) that start and finish on nodes in P_R, with associated semantics the composition of the semantics of the original edges $\tau_{e_1 \to \cdots \to e_n} = \tau_{e_n} \circ \cdots \circ \tau_{e_1}$. There are only a finite number of such edges, because the original graph is finite and removing P_R cuts all cycles. There may be several edges between two given nodes, because there may exist several control paths between these nodes in the original program. Equivalently, this multigraph can be obtained by starting from the original graph (P, E) and by removing all nodes p in $P \setminus P_R$ as follows: each couple of edges e_1, from p_1 to p, and e_2, from p to p_2, is replaced by a single edge from p_1 to p_2 with semantics $\tau_{p_2} \circ \tau_{p_1}$.

Example of Section 1 (Cont'd) The reduced control flow graph obtained for our running example is



Our analysis algorithm performs chaotic iterations over that reduced multigraph, without ever constructing it explicitly. We start from an iteration strategy, that is, a method for choosing which of the equations to apply next; one may for instance take a variant of the naive "breadth-first" algorithm from §2, but any iteration strategy [9, §3.7] befits us (see also Alg. 1). An iteration strategy maintains a set of "active nodes", which initially contains all nodes p such that $I_p \neq \emptyset$. It picks one edge e from an active node p_1 to a node p_2, and applies $X_{p_2} := X_{p_2} \sqcup \tau_e^\sharp(X_{p_1})$ in the case of a node $p_2 \in P_R \setminus P_W$, and applies $X_{p_2} := X_{p_2} \triangledown (X_{p_2} \sqcup \tau_e^\sharp(X_{p_1}))$ if $p_2 \in P_W$; then p_2 is added to the set of active nodes if the value of X_{p_2} has changed.

Our alteration to this algorithm is that we only pick edges e from p_1 to p_2 such that there exist $x_1 \in X_{p_1}$, $x_2 \in \tau_e(\{x_1\})$ and $x_2 \notin X_{p_2}$ with the current values of X_{p_1} and X_{p_2}. In other words, going back to the original control flow graph, we only pick paths that add new reachable states to their end node, and we temporarily *focus* on such a path.

How do we find such edges e out of potentially exponentially many? We express them as the solution of a *bounded reachability* problem — how can we go from control state p_1 with variable state in X_{p_1} to control state p_2 with variable state in X_{p_2} —, which we solve using satisfiability modulo theory (SMT). (See Alg. 2)

3.2 Finding Focus Paths

We now make the assumption that both the program transition semantics τ_e and the abstract elements $x^\sharp \in D$ can be expressed within a decidable theory T (this assumption may be relaxed by replacing the concrete semantics, including e.g. multiplicative arithmetic, by a more abstract one through e.g. linearization [30]).

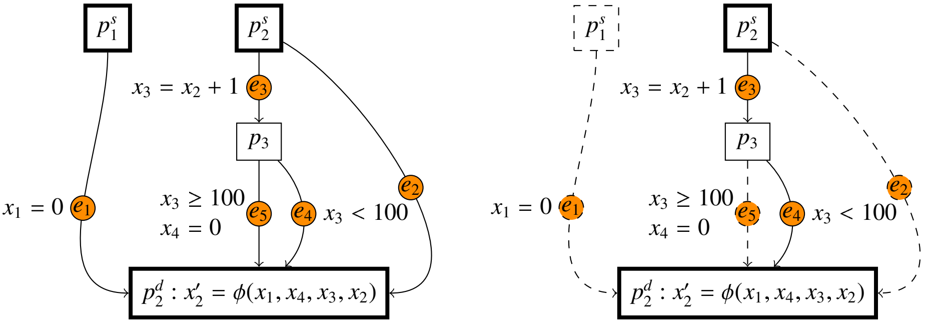
Such is for instance the case if the program operates on rational values, so a program state is an element of $\Sigma = \mathbb{Q}^n$, all operations in the program, including guards and assignments, are linear arithmetic, and the abstract domain is the domain of convex polyhedra over $\mathbb{Q}^n$, in which case T can be the theory of linear real arithmetic (LRA). If program variables are integer, with program state space $\Sigma = \mathbb{Z}^n$, but still retaining the abstract domain of convex polyhedra over $\mathbb{Q}^n$, then we can take T to be the theory of linear integer arithmetic (LIA). Deciding the satisfiability of quantifier-free formulas in either LIA or LRA, with atoms consisting in propositional variables and in linear (in)equalities with integer coefficients, is NP-complete. There however exist efficient decision procedures for such formulas, known as SMT-solvers, as well as standardised theories and file formats [6]; notable examples of SMT-solvers capable of dealing with LIA and LRA are Z3 and Yices. Kroening & Strichman [29] give a good introduction to the techniques and algorithms in SMT solvers.

We assume that the program is expressed in SSA form, with each program variable being assigned a value at only a single point within the program [18]; standard techniques exist for converting to SSA. Figure 1 gives both "normal" and SSA-form control-flow graphs for Listing 2.

We transform the original control flow graph (P, E) in SSA form by disconnecting the nodes in P_R: each node p_r in P_R is split into a "source" node p_r^s with only outbound edges, and a "destination" node p_r^d with only inbound edges. We call the resulting graph (P', E'). Figure 2(a) gives the disconnected SSA form graph for Listing 2 where p_1 and p_2 have been split.

We consider execution traces starting from a p_r^s node and ending in a p_r^d node. We define them as for doing bounded model checking [2]. To each node $p \in P'$ we attach a Boolean b_p or *reachability predicate*, expressing that the trace goes through program point p. For nodes p' not of the form p_r^s, we have a constraint $b_{p'} = \bigvee_p e_{p,p'}$, for $e_{p,p'}$ ranging over all incoming edges. To each edge $p \to p'$ we attach a Boolean $e_{p,p'}$, and a constraint $e_{p,p'} = b_p \wedge \tau_{p,p'}$. The conjunction ρ of all these constraints, expresses the transition relation between the p_r^s and p_r^d nodes (with implicit existential quantification).

If the transitions $\tau_{(p,p')}$ are non-deterministic, a little care must be exercised for the path obtained from the b_p to be unique. For instance, if from program point p_1 one can move non-deterministically to p_2 or p_3 through edges e_2 and e_3 an incorrect way of writing the formula would be $(b_2 = e_2) \wedge (b_3 = e_3) \wedge (e_2 = b_1) \wedge (e_3 = b_1)$, in which case b_2 and b_3 could be simultaneously true. Instead, we introduce special "choice" variables c_i that model non-deterministic choices (Fig. 2).



(a) Disconnected (SSA) CFG

(b) With a focus path (solid edges) from $x_2 = 0$ at program point 2 to $x_2' = 1$ at the same program point

$$(e_1 = (x_1 = 0) \wedge b_1^s) \wedge (e_3 = (x_3 = x_2 + 1) \wedge b_2^s \wedge c_2^s) \wedge (e_2 = b_2^s \wedge \neg c_2^s) \wedge (e_5 = b_3 \wedge x_3 \geq 100 \wedge x_4 = 0) \wedge (e_4 = b_3 \wedge x_3 < 100) \wedge (b_3 = e_3) \wedge (b_2^d = e_1 \vee e_4 \vee e_5 \vee e_2) \wedge (x_2' = ite(e_1, x_1, ite(e_5, x_4, ite(e_4, x_3, x_2))))$$

Fig. 2. Disconnected version of the SSA control flow graph of Fig. 1(b), and the corresponding SMT formula. $ite(b, e_1, e_2)$ is a SMT construct whose value is "if b then the value of e_1 else the value of e_2". To each node p_x corresponds a Boolean b_x and an optional choice variable c_x; to each edge, a Boolean e_y.

In order to find a path from program point $p_1 \in P_R$, with variable state x_1, to program point $p_2 \in P_R$, with variable state x_2, we simply conjoin ρ with the formulas $x_1 \in X_{p_1}$ and $x_2 \notin X_{p_2}$, with x_1, x_2, $x_1 \in X_{p_1}$ and $x_2 \notin X_{p_2}$ expressed in terms of the SSA variables.[3] For instance, if X_{p_1} and X_{p_2} are convex polyhedra defined by systems of linear inequalities, one simply writes these inequalities using the names of the SSA-variables at program points p_1 and p_2.

[3] The formula defining the set of values represented by an abstract element X has sometimes been denoted by $\hat{\gamma}$ [34].

We apply SMT-solving over that formula. The result is either "unsatisfiable", in which case there is no path from p_1, with variable values x_1, to p_2, with variable values x_2, such that $x_1 \in X_{p_1}$ and $x_2 \notin X_{p_2}$, or "satisfiable", in which case SMT-solving also provides a model of the formula (a satisfying assignment of its free variables); from this model we easily obtain such a path, unique by construction of ρ.

Indeed, a model of this formula yields a trace of execution: those b_p predicates that are true designate the program points through which the trace goes, and the other variables give the values of the program variables.

Example of Section 1 (Cont'd) The SSA form of the control flow graph of Figure 1(a) is depicted in Figure 1(b). Fig. 2 shows the disconnected version of the SSA Graph (the node p_2 is now split), and the formula ρ expressing the semantics is shown beneath it.

Then, consider the problem of finding a path starting in control point 2 inside polyhedron $x = 0$ and ending at the same control point but outside of that polyhedron. Note that because there are two outgoing transitions from node p_2^s, which are chosen non-deterministically, we had to introduce a Boolean choice variable c_2^s.

The focus path of Fig. 2(b) was obtained by solving the formula $\rho \wedge b_1^s = \text{false} \wedge b_2^s = \text{true} \wedge b_2^d = \text{true} \wedge (x_2 = 0) \wedge \neg(x_2' = 0)$: we impose that the path starts at point p_2^s (thus forcing $b_1^s = \text{false} \wedge b_2^s = \text{true}$) in the polyhedron $x = 0$ (thus $x_2 = 0$) and ends at point p_2^d (thus forcing $b_2^p = \text{true}$) outside of that polyhedron (thus $\neg(x_2 = 0)$).

3.3 Algorithm

Algorithm 2 consists in the iteration of the path finding method of Sec. 3.2, coupled with forward abstract interpretation along the paths found and, optionally, path acceleration.

3.4 Correctness and Termination

We shall now prove that this algorithm terminates, and that the resulting X_p define an inductive invariant that contains all initial states I_p. The proof is a variant of the correctness proof of the chaotic iterations.

The invariant maintained by this algorithm is that all nodes $p_1 \in P_R \setminus A$ are such that there is no execution trace starting at point p_1 in a state $x_1 \in X_{p_1}$ and ending at point p_2 in a state $x_2 \notin X_{p_2}$. Evidently, if A becomes empty, then this condition means that X_p is an inductive invariant.

Termination is ensured by the classical argument of termination of chaotic iterations in the presence of widening: they always terminate if all cycles in the control flow graph are broken by widening points [13, Th. 4.1.2.0.6, p. 128]. In short, an infinite iteration sequence is bound to select at least one node p in P_W an infinite amount of times, because P_W breaks all cycles, but due to the properties of widening, X_p should be stationary, which contradicts the infinite number of selections. Our comment at line 20 of Alg. 2 is important for termination: it ensures that for any widening node p, the sequence of values taken by X_p when it is updated and reinserted into set A is strictly ascending, which ensures termination in finite time.

Algorithm 2. Path-focused Algorithm

```
Compute SSA-form of the control flow graph.
Choose $P_R$, compute the disconnected graph $(P', E')$ accordingly.
$\rho \leftarrow$ computeFormula$(P', E')$                                  ▹ Precomputations
$A \leftarrow \emptyset$;
for all $p \in P_R$ such that $I_p \neq \emptyset$ do
    $A \leftarrow A \cup \{p\}$
end for;
while $A$ is not empty do                         ▹ Fixpoint Iteration on the reduced graph
    Choose $p_1 \in A$
    $A \leftarrow A \setminus \{p_1\}$
    repeat
        $res \leftarrow \text{SmtSolve}\left(\rho \wedge b_{p_1} \wedge x_1 \in X_{p_1} \wedge \bigvee_{p_2 | (p_1, p_2) \in E'} \left(b_{p_2} \wedge x_2 \notin X_{p_2}\right)\right)$
        if $res$ is not "unsat" then
            Compute $e' \in E'$ from $res$          ▹ Extraction of path from the model (§3.2)
            $Y \leftarrow \tau^{\sharp}_{e'}(X_{p_1})$
            if $p_2 \in P_W$ then
                $X_{temp} \leftarrow X_{p_2} \triangledown (X_{p_2} \sqcup Y)$      ▹ Final point $p_2$ is a widening point
            else
                $X_{temp} \leftarrow X_{p_2} \sqcup Y$
            end if
                        ▹ at this point $X_{temp}$   $X_{p_2}$ otherwise $p_2$ would not have been chosen
            $X_{p_2} \leftarrow X_{temp}$
            $A \leftarrow A \cup \{p_2\}$
        end if
    until $res$="unsat"
end while                                                      ▹ End of Iteration
Possibly narrow (see Sec. 4.1)
Compute $X_{p_i}$ for $p_i \notin P_R$
return all $X_{p_i}$
```

3.5 Self-Loops

The algorithm in the preceding subsection is merely a "clever" implementation of standard polyhedral analysis [17,27] on the reduced control multigraph (P_R, E_R); the difference with a naive implementation is that we do not have to explicitly enumerate an exponential number of paths and instead leave the choice of the focus path to the SMT-solver. We shall now describe an improvement in the case of self-loops, that is, single paths from one node to itself.

Algorithm 3 is a variant of Alg. 2 where self-loops are treated specially:

- The *loopiter*$(\tau^{\sharp}, X)$ function returns the result of a widening / narrowing iteration sequence for abstract transformer $\tau^{\sharp}$ starting in X; it returns X' such that $X \subseteq X'$ and $\tau^{\sharp}(X') \subseteq X'$.
- In order not to waste the precision gained by *loopiter*, the first time we consider a self-loop e' we apply a union operation instead of a widening; set U records the self-loops that have already been visited. This is a form of delayed widening [28].

Algorithm 3. Path-focused Algorithm with Self-Loops. 🟧 marks changes from Alg. 2.

```
Compute SSA-form of the control flow graph.
Choose $P_R$, compute the disconnected graph $(P', E')$ accordingly.
$\rho \leftarrow$ computeFormula$(P', E')$                         ▷ Precomputations
$A \leftarrow \emptyset$;
for all $p \in P_R$ such that $I_p \neq \emptyset$ do
    $A \leftarrow A \cup \{p\}$
end for;
while $A$ is not empty do                     ▷ Fixpoint Iteration on the reduced graph
    Choose $p_1 \in A$
    $A \leftarrow A \setminus \{p_1\}$
    $U = \emptyset$                                ▷ $U$ is a set of "already seen" edges
    repeat
        $res \leftarrow \text{SmtSolve}\left(\rho \wedge b_{p_1} \wedge x_1 \in X_{p_1} \wedge \bigvee_{p_2 | (p_1, p_2) \in E'} \left(b_{p_2} \wedge x_2 \notin X_{p_2}\right)\right)$
        if $res$ is not "unsat" then
            Compute $e' \in E'$ from $res$
            if $p_1 = p_2$ then
                $Y \leftarrow loopiter(\tau^{\sharp}_{e'}, X_{p_1})$
            else
                $Y \leftarrow \tau^{\sharp}_{e'}(X_{p_1})$
            end if
            if $p_2 \in P_W$ and $(p_1 \neq p_2 \vee e' \in U)$ then
                $X_{p_2} \leftarrow X_{p_2} \nabla (X_{p_2} \sqcup Y)$       ▷ Final point $p_2$ is a widening point
            else
                $X_{p_2} \leftarrow X_{p_2} \sqcup Y$
                $U \leftarrow U \cup \{e'\}$
            end if
            $A \leftarrow A \cup \{p_2\}$
        end if
    until $res$="unsat"
end while                                                  ▷ End of Iteration
Compute $X_{p_i}$s for $p_i \notin P_R$
return all $X_{p_i}$s
```

Termination is still guaranteed, because the inner loop cannot loop forever: it can visit any self-loop edge e' at most once before applying widening.

Example of Section 1 (Cont'd) Let us perform our algorithm on our example :

- Step 1 : Is there a path from control point p_1 to control point p_2 feasible (without additional constraint) ? Yes. On Figure 2, the obtained model corresponds to the transition from p_1^s to p_2^d, and leads to the interval $X_{p_2} = [0, 0]$.
- Step 2 : Is there a path from p_2 with $x = 0$ to p_2 with $x \neq 0$? The answer to this query is depicted in Figure 2(b): there is such a path, on which we now focus. This path is considered as a loop and we therefore do a local iteration with widenings

(*loopiter*). X_{p_2} becomes $[0, 1]$, then after widening $[0, \infty]$. A narrowing step gives finally $X_{p_2} = [0, 99]$, which is thus the result of *loopiter*.
- Step 3 : Is there a path from p_2 with $x \in [0, 99]$ to p_2 with $x' \notin [0, 99]$? No.

The iteration thus ends with the desired invariant.

4 Extensions

4.1 Narrowing

Narrowing iterations can also be applied within our framework. Let us assume that some inductive invariant $X_{p \in P_R}$ has been computed; it satisfies the relation $\psi(X) \subseteq X$ component-wise, noting $X = (X_1, \dots, X_{|P|})$, and $\psi(X)$ denotes $(Y_1, \dots, Y_{|P|})$ defined as

$$Y_{p_2} = I_{p_2} \cup \bigcup_{e \in E_R \ e \text{ from } p_1 \text{ to } p_2} \tau_e\left(X_{p_1}\right) \tag{4}$$

The abstract counterpart to this operator is $\psi^\sharp$, defined similarly, replacing τ by $\tau^\sharp$ and $\cup$ by $\sqcup$. It satisfies the correctness condition (see Rel. 1) $\forall X \in D \ \psi(X) \subseteq \psi^\sharp(X)$.

As per the usual narrowing iterations, we compute a narrowing sequence $X^{(k)} = {\psi^\sharp}^k(X)$. It is often sufficient to stop at $k = 1$; otherwise one may stop when $X^{(k+1)} \nsubseteq X^{(k)}$. Let us now see a practical algorithm for computing $Y = \psi^\sharp(X)$:

For all $p \in P_R$, we initialise $Y_p := I_p$. For all $p_2 \in P_R$, we consider all paths $e \in E_R$ from $p_1 \in P_R$ to p_2 such that there exist $x_1 \in X_{p_1}$, $x_2 \in X_{p_2}$, $x_2 \in \tau_e(\{x_1\})$ as explained in §3.2. We then update $Y_{p_2} := Y_{p_2} \sqcup \tau^\sharp_e(X_{p_1})$.

4.2 Acceleration

In Sec. 3.5, we have described *loopiter* function that performs a classical widening / narrowing iteration over a single path. In fact, the only requirement over it is that *loopiter*$(\tau^\sharp, X)$ returns X' such that $X \subseteq X'$ and $\tau^\sharp(X') \subseteq X'$. In other words, X' is an over-approximation of ${\tau^\sharp}^*(X)$, noting R^* the transitive closure of R.

In some cases, we can compute directly such an over-approximation, sometimes even obtaining ${\tau^\sharp}^*(X)$ exactly; this is known as *acceleration* of the loop. Examples of possible accelerations include the case where τ_e is given by a difference bound matrix [12], an octagon [10], ultimately periodic integer relations [11] or certain affine linear relations [23,22,1].

For instance, the focus path of Fig. 2(b) consists in the operations and guards $x = x + 1; x < 100$; instead of iterating that path, we can compute its exact acceleration, yielding $x \in [0, 99]$.

4.3 Partitioning

It is possible to partition the states at a given program point according to some predicate or a partial history of the computation [36]. This amounts to introducing several graph nodes representing the same program point, and altering the transition relation.

4.4 Input-Output Relations

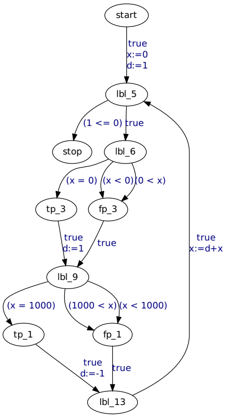
As with other analyses using relational domains, it is possible to obtain abstractions of the input-output relation of a program block or procedure instead of an abstraction of the set of states at the current point [1]; this also allows analyzing recursive procedures [27, Sec. 7.2]. It suffices to include in the set of variables copies of the variables at the beginning of the block or procedure; then the abstract value obtained at the end of the block or procedure is the desired abstraction.

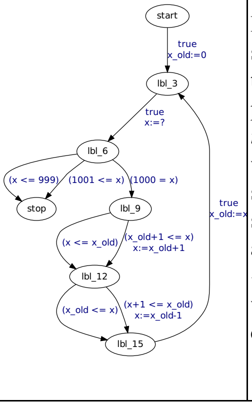
5 Implementation and Preliminary Results

Our algorithm has been implemented as an option for Aspic, that computes invariants from counter automata with Linear Relation Analysis ([20]). We wrote an Ocaml interface to the Yices SMT-solver ([19]), and modified the fixpoint computation inside

Table 1. Invariant generation on two simple challenging programs

Program	Automaton	Result and notes

Listing 4. Boustrophedon
`void boustrophedon() {`
`  int x;`
`  int d;`
`  x = 0;`
`  d = 1;`
`  while (1) {`
`    if (x == 0) d=1;`
`    if (x == 1000) d=-1;`
`    x += d;`
`  }`
`}`
	 start
	true
	x:=0
	d:=1
	lbl_5
	(1 <= 0)
	true
	stop
	lbl_6
	(x = 0)
	(x < 0)
	(0 < x)
	tp_3
	fp_3
	true
	d:=1
	true
	true
	x:=d+x
	lbl_9
	(x = 1000)
	(1000 < x)
	(x < 1000)
	tp_1
	fp_1
	true
	d:=-1
	true
	lbl_13
		The compilation of the program gives an expanded control structure where some paths are "clearly" unfeasible (e.g. imposing both $x < 0$ and $x > 1000$), thus the only feasible ones are guarded by $x < 0$, $x = 0$, $0 < x < 1000$, $x = 1000$ and $x > 1000$.
		The tool finds the invariant $\{\mathbf{0 \le x \le 1000, -1 \le d \le 1}\}$ Classical Analysis with widening "upto" gives $\{d \le 1, d + 1999 \ge 2x\}$ and Gopan and Reps' improvement is not able to find $x \ge 0$.

Listing 5. Rate limiter
`void main() {`
`  float x_old, x;`
`  x_old = 0;`
`  while (1) {`
`    x = input(-1000,1000);`
`    if (x >= x_old+1)`
`        x = x_old+1;`
`    if (x <= x_old-1)`
`        x = x_old-1;`
`    x_old = x;`
`  }`
`}`
Source : [32]
	 start
	true
	x_old:=0
	lbl_3
	true
	x:=?
	lbl_6
	(x <= 999)
	(1001 <= x)
	(1000 = x)
	stop
	lbl_9
	true
	x_old:=x
	(x <= x_old)
	(x_old+1 <= x)
	x:=x_old+1
	lbl_12
	(x_old <= x)
	(x+1 <= x_old)
	x:=x_old-1
	lbl_15
		In order to properly analyse such a program, Astrée distinguishes all four execution paths inside the loop through *trace partitioning* [36], which is triggered by ad hoc syntactic criteria (e.g. two successive if-then-else). Our algorithm finds the invariant $\{\mathbf{-1000 \le x_{old} \le 1000}\}$, which is not found by classical analysis.

Aspic to deal with local iterations of paths. The implementation still needs some improvements, but the preliminary results are promising, and we describe some of them in Table 1. We provide no timing results since we were unable to detect any overcost due to the method. These two examples show that since we avoid (some) convex hulls, the precision of the whole analysis is improved.

The rate limiter example is particularly interesting, since, like the one in Listing 1 (which does not include a loop), it will be imprecisely analyzed by any method enforcing convex invariants at intermediate steps.

6 Related Work

Our algorithm may be understood as a form of *chaotic iterations* [13, §2.9.1, p. 53] over a certain system of semantic questions; we use SMT as an oracle to know which equations need propagating. The choice of widening points, and the order in which to solve the abstract equations, have an impact on the precision of the whole analysis, as well as its running time. Even though there exist few hard general results as to which strategy is best [13, §4.1.2, p. 125], some methods tend to experimentally behave better [9].

"Lookahead widening" [24] was our main source of inspiration: iterations and widenings are adapted according to the discovery of new feasible paths in the program. This approach avoids loss of precision due to widening in programs with multiple paths inside loops. It has proved its efficacy to suppress some gross over-approximations induced by naive widening. However, it does not solve the imprecisions introduced by convex hull (e.g. it produces false alarms on Listing 1).

Our method analyzes separately the paths between cut-nodes. We have pointed out that this is (almost) equivalent to considering finite unions of elements of the abstract domain, known as the *finite powerset* construction, between the cut-nodes.[4] The finite powerset construction is however costly even for loop-free code, and it is not so easy to come up with widening operators to apply it to codes with loops or recursive functions [4]; for limiting the number of elements in the unions, some may be lumped together (thus generally introducing further over-approximation) according to affinity heuristics [37,33].

Still, in the recent years, much effort has been put into the discovery of *disjunctive invariants*, for instance in predicate abstraction [25]. Of particular note is the recent work by Gulwani and Zuleger on inferring disjunctive invariants [26] for finding bounds on the number of iterations of loops. We improve on their method on two points:

- In contrast to us, they assume that the transition relation is given in disjunctive normal form [26, Def. 5], which in general has exponential size in the number of tests inside the loop. By using SMT-solving, we keep the DNF implicit and thus avoid this blowup.

[4] It is equivalent if the only source of disjunctions are the splits in the control flow, and not atomic operations. For instance, if the test $|x| \geq 1$ is considered an atomic operation, then we could take the disjunction $x \geq 1 \vee x \leq -1$ as output. We can rephrase that as a control flow problem by adding a test $x \geq 0$, otherwise said to express $|x|$ as a piecewise linear function with explicit tests for splits between the pieces.

- By using acceleration, we may obtain more precise results than using widening, as they do for lattices that do not satisfy the ascending chain condition.

Nevertheless, their method allows expressing disjunctive invariants at loop heads, and not only at intermediate points, as we do. However, we think it is possible to get the best of both worlds and combine our method with theirs. In order to obtain a disjunctive invariant, they first choose a "convexity witness" (given that the number of possible witnesses is exponential, they choose it using heuristics) [26, p. 7], and then they compute a "transitive closure" [26, Fig. 6], which is a form of fixed point iteration of input-output relations (as in our Sec. 4.4) over an expanded control-flow graph. The choice of the convexity witness amounts to a partitioning of the nodes and transition (Sec. 4.3). Thus, it seems to possible to apply their technique, but replace their fixed point iteration [26, Fig. 6] by one based on SMT-solving and path focusing, using acceleration if possible.

In recent years, because of improvement in SMT-solving, techniques such as ours, distinguishing *paths* inside loops, have become tractable [31,7,32,21]. An alternative to using SMT-solving is to limit the number and length of traces to consider, as in *trace partitioning* [36], used in the Astrée analyzer [16,15,8], but the criteria for limitation tend to be ad hoc. In addition, methods for abstracting the sets of paths inside a loop, weeding out infeasible paths, have been introduced [5].

With respect to optimality of the results, our method will generate the strongest inductive invariant inside the abstract domain if the domain satisfies the ascending chain condition and no widening is used; for other domains, like all methods using widenings, it may or may not generate it. In contrast, some recent works [21] guarantee to obtain the strongest invariant for the same analysis problem, at the expense of restriction to template linear domains and linear constructions inside the code.

7 Conclusion and Future Work

We have described a technique which leverages the bounded model checking capacities of current SMT solvers for guiding the iterations of an abstract interpreter. Instead of normal iterations, which "push" abstract values along control-flow edges, including control-flow splits and merges, we consider individual paths. This enables us, for instance, to use acceleration techniques that are not available when the program fragment being considered contains control-flow merges. This technique computes exact least invariants on some examples on which more conventional static analyzers incur gross imprecision or have to resort to syntactic heuristics in order to conserve precision.

We have focused on numerical abstractions. Yet, one would like to use similar techniques for heap abstractions, for instance. The challenge will then be to use a decidable logic and an abstract domain such that both the semantics of the program statements and the abstract values can be expressed in this logic. This is one direction to explore. With respect to the partitioning technique, 4.3, we currently express the partition as multiple explicit control nodes, but it seems desirable, for large partitions (e.g. according to Boolean values, as in B. Jeannet's BDD-Apron library) to represent them succinctly; this seems to fit nicely with our succinct encoding of the transition relation as a SMT-formula.

Another direction is to evaluate the scalability of these methods on larger programs. The implementation needs to be tested more to evaluate the precision of our method on middle-sized programs, the main advantage is that Aspic implements some of the acceleration techniques. Analyzers such as Astrée scale up to programs running a control loop several hundreds of thousands of lines long; translating such a loop to a SMT formula and solving for this formula and additional constraints does not seem tractable. It is possible that semantic slicing techniques [35] could help in reducing the size of the generated SMT problems.

References

1. Ancourt, C., Coelho, F., Irigoin, F.: A modular static analysis approach to affine loop invariants detection. Electr. Notes Theor. Comput. Sci. 267(1), 3–16 (2010); Proceedings of NSAD
2. Armando, A., Mantovani, J., Platania, L.: Bounded model checking of software using smt solvers instead of sat solvers. International Journal on Software Tools for Technology Transfer (STTT) 11(1), 69–83 (2009)
3. Bagnara, R., Hill, P.M., Zaffanella, E.: The Parma Polyhedra Library, version 0.9, http://www.cs.unipr.it/ppl
4. Bagnara, R., Hill, P.M., Zaffanella, E.: Widening operators for powerset domains. International Journal on Software Tools for Technology Transfer (STTT) 8(4-5), 449–466 (2006); See also erratum in June 2007 issue
5. Balakrishnan, G., Sankaranarayanan, S., Ivancic, F., Gupta, A.: Refining the control structure of loops using static analysis. In: Chakraborty, S., Halbwachs, N. (eds.) EMSOFT, pp. 49–58. ACM, New York (2009)
6. Barrett, C., Ranise, S., Stump, A., Tinelli, C.: The satisfiability modulo theories library, SMT-LIB (2008), www.smtlib.org
7. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Path invariants. In: PLDI, pp. 300–309. ACM, New York (2007)
8. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Programming Language Design and Implementation (PLDI), pp. 196–207. ACM, New York (2003)
9. Bourdoncle, F.: Sémantique des langages impératifs d'ordre supérieur et interprétation abstraite. Ph.D. thesis, École polytechnique, Palaiseau (1992)
10. Bozga, M., Gîrlea, C., Iosif, R.: Iterating octagons. Tech. Rep. 16, VERIMAG (2008)
11. Bozga, M., Iosif, R., Konecny, F.: Fast acceleration of ultimately periodic relations. Tech. Rep. 2010-3, VERIMAG (2010)
12. Comon, H., Jurski, Y.: Multiple counters automata, safety analysis and Presburger arithmetic. In: Hu, A.J., Vardi, M.Y. (eds.) CAV 1998. LNCS, vol. 1427, pp. 268–279. Springer, Heidelberg (1998)
13. Cousot, P.: Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes. State doctorate thesis, Université scientifique et médicale de Grenoble and Institut National Polytechnique de Grenoble (1978), http://tel.archives-ouvertes.fr/tel-00288657/en/
14. Cousot, P., Cousot, R.: Abstract interpretation frameworks. J. of Logic and Computation, 511–547 (August 1992)
15. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTREÉ analyzer. In: Sagiv, S.M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 21–30. Springer, Heidelberg (2005)

16. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Combination of abstractions in the ASTRÉE static analyzer. In: Okada, M., Satoh, I. (eds.) ASIAN 2006. LNCS, vol. 4435, pp. 272–300. Springer, Heidelberg (2008), http://www.di.ens.fr/~cousot/COUSOTpapers/ASIAN06.shtml
17. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Principles of Programming Languages (POPL), pp. 84–96. ACM, New York (1978)
18. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: An efficient method of computing static single assignment form. In: Principles of Programming Languages (POPL), pp. 25–35. ACM, New York (1989)
19. Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
20. Feautrier, P., Gonnord, L.: Accelerated invariant generation for C programs with Aspic and C2fsm. In: Tools (TAPAS) (2010)
21. Gawlitza, T.M., Monniaux, D.: Improving strategies via SMT solving. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 236–255. Springer, Heidelberg (2011)
22. Gonnord, L.: Accélération abstraite pour l'amélioration de la précision en analyse des relations linéaires. Ph.D. thesis, Université Joseph Fourier (October 2007), http://tel.archives-ouvertes.fr/tel-00196899/en/
23. Gonnord, L., Halbwachs, N.: Combining widening and acceleration in linear relation analysis. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 144–160. Springer, Heidelberg (2006)
24. Gopan, D., Reps, T.W.: Lookahead widening. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 452–466. Springer, Heidelberg (2006)
25. Gulwani, S., Srivastava, S., Venkatesan, R.: Constraint-based invariant inference over predicate abstraction. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 120–135. Springer, Heidelberg (2009)
26. Gulwani, S., Zuleger, F.: The reachability-bound problem. In: Zorn, B.G., Aiken, A. (eds.) PLDI, pp. 292–304. ACM, New York (2010)
27. Halbwachs, N.: Détermination automatique de relations linéaires vérifiées par les variables d'un programme. State doctorate thesis, Université scientifique et médicale de Grenoble and Institut National Polytechnique de Grenoble (1979), http://tel.archives-ouvertes.fr/tel-00288805/en/
28. Halbwachs, N.: Delay analysis in synchronous programs. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 333–346. Springer, Heidelberg (1993)
29. Kroening, D., Strichman, O.: Decision procedures. Springer, Heidelberg (2008)
30. Miné, A.: Symbolic methods to enhance the precision of numerical abstract domains. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 348–363. Springer, Heidelberg (2005)
31. Monniaux, D.: Automatic modular abstractions for linear constraints. In: Pierce, B.C. (ed.) Symposium on Principles of Programming Languages (POPL). ACM, New York (2009)
32. Monniaux, D.: Automatic modular abstractions for template numerical constraints. Logical Methods in Computer Science (2010) (to appear)
33. Popeea, C., Chin, W.N.: Inferring disjunctive postconditions. In: Okada, M., Satoh, I. (eds.) ASIAN 2006. LNCS, vol. 4435, pp. 331–345. Springer, Heidelberg (2008)
34. Reps, T.W., Sagiv, M., Yorsh, G.: Symbolic implementation of the best transformer. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 252–266. Springer, Heidelberg (2004)
35. Rival, X.: Understanding the origin of alarms in ASTRÉE. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 303–319. Springer, Heidelberg (2005)
36. Rival, X., Mauborgne, L.: The trace partitioning abstract domain. Transactions on Programming Languages and Systems (TOPLAS) 29(5), 26 (2007)
37. Sankaranarayanan, S., Ivančić, F., Shlyakhter, I., Gupta, A.: Static analysis in disjunctive numerical domains. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 3–17. Springer, Heidelberg (2006)

Author Index